

Courant Computer Science Report #10

November 1976

NYU NOS-10

A Hierarchical Technique for Mechanical Theorem Proving and Its Application to Programming Language Design

Norman Rubin

Courant Institute of
Mathematical Sciences
Computer Science Department



New York University

Report No. NOS-10 prepared under Contract
No. N00014-75-C-0571
with the Office of Naval Research

COURANT COMPUTER SCIENCE PUBLICATIONS

Price

COURANT COMPUTER SCIENCE NOTES

- Programming Languages and Their Compilers, J. Cocke
& J. T. Schwartz, 2nd Revised Version, *April 1970, iii+767 pp.* \$23.00
- On Programming: An Interim Report on the SETL Project. 20.50
Part I: Generalities
Part II: The SETL Language and Examples of Its Use
(Parts I and II are consolidated in this volume.)
J. T. Schwartz, Revised *June 1975, xii+675 pp.*
- A SETLB Primer. H. Mullish & M. Goldstein, 1973, *v+201 pp.* 6.25
- Combinatorial Algorithms. E. G. Whitehead, Jr., 1973, *vi+104 pp.* 3.25

COURANT COMPUTER SCIENCE REPORTS

- No. 1 ASL: A Proposed Variant of SETL
Henry Warren, Jr., 1973, 326 pp.
- No. 2 A Metalanguage for Expressing Grammatical Restrictions in Nodal Spans Parsing of Natural Language.
Jerry R. Hobbs, 1974, 266 pp.
- No. 3 Type Determination for Very High Level Languages
Aaron M. Tenenbaum, 1974, 171 pp.
- No. 4 A Comprehensive Survey of Parsing Algorithms for Programming Languages. Phillip Owens. *Forthcoming.*
- No. 5 Investigations in the Theory of Descriptive Complexity.
William L. Gewirtz, 1974, 60 pp.
- No. 6 Operating System Specification Using Very High Level Dictions.
Peter Markstein, 1975, 152 pp.
- No. 7 Directions in Artificial Intelligence: Natural Language Processing. Ed. Ralph Grishman, 1975, 107 pp.
- No. 8 A Survey of Syntactic Analysis Procedures for Natural Language.
Ralph Grishman, 1975, 94 pp.
- No. 9 Scene Analysis: A Survey.
Carl Weiman, 1975, 62 pp.
- No. 10 A Hierarchical Technique for Mechanical Theorem Proving and Its Application to Programming Language Design.
Norman Rubin, 1976, 172 pp.

A catalog of SETL Newsletters and other SETL-related material is also available. Courant Computer Science Reports are available upon request. Prepayment is required for Courant Computer Science Notes. Please address all communications to

COURANT INSTITUTE OF MATHEMATICAL SCIENCES
251 Mercer Street
New York, N. Y. 10012

COURANT INSTITUTE OF MATHEMATICAL SCIENCES

Computer Science

NSO-10

A HIERARCHICAL TECHNIQUE FOR MECHANICAL THEOREM PROVING
AND ITS APPLICATION TO PROGRAMMING LANGUAGE DESIGN

Norman Rubin

Report No. NOS-10 prepared under
Contract No. N00014-75-C-0571
with the Office of Naval Research

Table of Contents

<u>Section</u>	<u>Page</u>
1. <u>GENERAL INTRODUCTION AND REVIEW OF THE LITERATURE</u>	1
1.1 Declarative vs. Imperative Languages	5
1.2 Deductive Languages	7
1.3 Resolution	9
1.4 Languages Based on Formal Logic	10
2. <u>THE THEOREM PROVING SYSTEM</u>	13
2.1 Resolution Proofs	13
2.2 Refutation Graphs	14
2.3 Horn Clauses	20
2.4 The Revised Unification Problem	23
2.5 Equality	28
2.6 Equality-Generalized Resolution	31
2.7 Coercion	38
2.8 Comparison of Coercion and Resolution	44
3. <u>THE DILEMMA PROGRAMMING LANGUAGE</u>	49
3.1 Horn Clauses as Procedures	51
3.2 An Example	54
3.3 Syntax and Semantics	58
3.4 Partial Evaluation	66
3.5 Completions	74
3.6 How to Control the Search	76
3.7 Built-in Equality	77
3.8 System Parameters	77
3.9 Missionary and Cannibal Example	78
3.10 Implementation Outline	81
3.11 Syntax Summary	83
4. <u>THE GEOMETRY PROGRAM</u>	85
4.1 Basic Objects	90
4.2 Logical Axioms	91
4.3 Coercions	99
4.4 Strategies	100
4.5 Point Constructions	104
5. <u>EXPERIMENTAL RESULTS</u>	109
5.1 The G0 Problem	109
5.2 The G1 Problem	115
5.3 The G2 Problem	117
5.4 The G3 Problem	121
5.5 The G4 Problem	123
5.6 The G5 Problem	126
6. <u>CONCLUSIONS</u>	128
6.1 Formalistic Methods	128
6.2 Hierarchical Theorem Proving	128
6.3 Partial Evaluation	129
6.4 The Geometry Program	130
6.5 Future Possibilities	130
Appendix I. Implementation Details and Listing	134
Appendix II. Geometry Program Listing	159
Bibliography	168

Acknowledgements

I would like to acknowledge the assistance of Professor M. Harrison who guided the research. Also I would like to thank D. Yarmush, R. Gurkewitz and V. Di Gri for their many suggestions and criticisms of the ideas in this paper, and their helpful comments on previous drafts.

Computations were performed at the ERDA Mathematics and Computing Laboratory under the sponsorship of ERDA, Contract No. E(11-1)-3077.

Finally I would like to thank C. Engle for her excellent typing job.

CHAPTER 1

GENERAL INTRODUCTION AND REVIEW OF THE LITERATURE

The development of effective computer algorithms for general problem solving has long been considered one of the fundamental goals of Artificial Intelligence research. Within the AI community there appear to be two distinct points of view concerning the design of such algorithms. The first or "formalistic" view is based on the claim that problem solving is a special case of theorem proving. Here, a problem solving system begins by formulating the problem as the axioms of a logical system. Next a mechanical theorem prover is applied to produce a proof of the existence of a solution to the problem. Usually the techniques used to produce the proof are independent of the particular problem area. Finally the proof is translated back into a solution of the original problem.

This methodology for general problem solving is quite attractive. The development of algorithms may concentrate on the formal theorem prover rather than on the specific problem area, so each advance may contribute to problem solving as a whole. Theorem proving may be analyzed mathematically in terms of consistency, completeness, and adequacy. Eventually we may be able to compare the efficiency of two algorithms without comparing the efficiency of two implementations. Since the descriptions of these algorithms are precise mathematical statements, they can be easily communicated. Finally the meta-theorems which may be proved

are often elegant and interesting in their own right.

However, such an approach has so far proved to be extremely inefficient. Among the reasons for this are the following: Theorem provers do not use higher level planning, and their search procedures seem to lack structure. They do not make use of specialized information about the problem. Indeed, the very generality of uniform procedures seems to imply that information about specific problem domains cannot be included. Theorem provers almost always use a poor notation, such as, for example, n applications of the successor function to represent n . While we may formalize any fact within predicate calculus it is more difficult to formalize how to use that fact. Such information, which is sometimes called procedural knowledge, often cannot be given to a theorem prover at all. Finally theorem provers tend to get "lost" in what are minor aspects of a problem. For example most theorem provers find it difficult to show that 3 times 2 is 6; worse still, they find it more difficult to show that 3 times 2 is not 7.

The other point of view among AI researchers is called "intuitive". Here programs are designed to conform to ideas about how people solve particular problems. Usually these problems come from some toy domain. These intuitive systems are strong exactly where the formalistic systems are weak. That is they are efficient but obscure. For example, it is clear that Winograd's blocks world system [44] is an effective program; yet there is no simple explanation of exactly how it works.

Our own approach to problem solving has been between these two extremes but closer to the first. We began with a new theory of formal deduction which is based on a hierarchical system of cooperating theorem provers. Using a special case of this theory we developed a computer program which appears superior, on an initial set of test problems, to resolution-based schemes; we then extended the theorem proving system in two directions. The first of these, partial evaluation, allowed procedural knowledge, heuristics, and domain-dependent information to be given to the theorem prover. The second extension, completions, is a way of summarizing the partial proofs developed by a theorem prover with an eye toward producing useful lemmas. The theorem prover together with these extensions serves as the interpreter for a programming language, which we call Dilemma. This language is of very high level. The general objective underlying the design of Dilemma has been to provide a programming language in which all of the diverse heuristics of successful AI programs may be used in conjunction with the naturalness and clarity of the predicate calculus. Finally we wrote a program in Dilemma, which produces proofs for a small subset of geometry theorems. Experimental results show that this program can solve most of the geometry problems solved by intuitive AI systems; yet the program is much shorter.

Our experience suggests that the techniques used by Dilemma significantly extend the range of application of resolution-based systems. In some cases these techniques give a realistic alternative to the more ad hoc intuitive programs, with the advantage that programs written in Dilemma are much easier to write and to read and yet not much more time consuming for a machine to execute.

The remainder of this chapter is a historical review designed to show how our work is related to previous research. Chapter 2 develops the theorem proving system mathematically. Chapter 3 concentrates on the language. Chapter 4 describes the geometry program. Chapter 5 gives the experimental results which we obtained with the geometry program. Finally Chapter 6 gives our conclusions and some ideas on future work.

1.1 Declarative vs. Imperative Languages

In 1958, John McCarthy [25] proposed writing a program for solving problems by manipulating sentences in a formal language. He wished to have a program which would learn in a way similar to the way humans do. McCarthy argued that, in order for a program to be capable of learning something it must be capable of being told about it. Therefore, interesting ideas must be expressible in simple ways. To focus on simple ways to express ideas he looked at the difference between the form of language used to teach people and that used to teach machines. The difference was: a machine is instructed mainly in the form of a sequence of imperative sentences while a person is taught mainly in declarative sentences describing the situation in which action is required together with a few imperatives. Some tasks are given to people in the first form, e.g. cooking directions, how to build a model airplane, how to knit. However, complex tasks such as how to prove a theorem of mathematics are taught declaratively. McCarthy's distinction between declarative and imperative languages is basic to our work. Essentially, our view is that declarative languages do not contain flow-of-control statements. They contain definitions, concepts and facts. Any of these may be applied at any place. On the other hand, imperative languages have DO loops, GO TOs, WHILEs, and other control dictions, all of which serve to specify what to do next.

1.1.1 Advantages of Declarative Languages

- (1) Previous knowledge can be used. A machine given a declarative sentence can integrate it into all the previous declarations.
- (2) Declarative sentences have logical consequences. Ideally, the machine could automatically deduce for itself the consequences of what it already knows.
- (3) Since the declarative sentences may be accessed in any order their meaning is much less dependent on the order that they are given to a machine. This can facilitate later improvements (or afterthoughts).
- (4) The effects of a new declarative are less dependent on the previous state of the system. When a change is made to an imperative system it generally requires much detailed knowledge about the system. In declaratives, changes may be made knowing less.
- (5) Declarative statements are easier for people to understand. A long program in an imperative form is usually difficult for a person to follow. He must first learn the global flow of the program and then the detailed interactions of each part. Parts of declarative programs on the other hand may be understood in isolation.

1.1.2 Advantages of Imperative Languages

- (1) Imperative programs are faster, since the flow of control is explicit and no search is required.
- (2) Computers execute imperative instructions.
The problem of writing a compiler or interpreter is simplified since the source and target languages are of similar form.
- (3) When it is clear how to do something, it is easier to state it as an imperative; when we try to encode a sequence of commands as declaratives we have to use unnatural tricky constructions, such as state variables.
- (4) The search strategies can be explicit in imperative forms. If we do not know the correct search strategy then we do not really know how to solve the problem. In this case the use of declaratives just hides our ignorance.

1.2 Deductive Languages

The fact that imperative programs are faster has often been considered to outweigh all the advantages of declarative programs. Almost all computer languages have been imperative. Minsky [26] in his review of AI felt that languages of the type we are calling declarative could best be called "descriptive". This was based on the view that each program statement would represent a high level abstract description of a situation. Hewitt [20] offered the name "procedural" for the class of languages we call imperative.

He claimed that knowlege could best be represented in a machine in the form of statements of a language which could be directly executed, e.g., a definition would be a list of properties and the code to actually test if an item had these properties.

Some three years after his first paper, McCarthy [25] gave a sketch of how his program would operate. He stated that a logical system more or less in the style of first order predicate calculus would be the right form for a declarative language.

First order logic has several attractive features for a declarative language. First it is well understood. Next, many people know it. It is clear that people can use it. Finally its mathematical structure has been well studied.

McCarthy envisaged a system in which first order predicate calculus statements would be read in, and some sort of deduction scheme (possibly modus ponens) would be used to deduce consequences until the answer came out.

Deduction is the central problem in using predicate calculus as a programming language. How could logical inference be set up on a computer? In 1964 several researchers attempted to build systems which used predicate calculus as an internal form. In Raphael's [29] SIR program, he allowed six predicate relations and wrote separate procedures for deduction

from each pair. Black [3] wrote a question answering system using several inference rules, each of which was a restricted form of modus ponens using a weak form of unification rule. Experience with these systems showed that they were hard to use and harder to extend. For example to add a new predicate to Raphael's program one had to be very familiar with the internals, and had to write a procedure specifying just how this predicate was related to all of the others. Cordell Green developed the QAL question answering system [14] as another limited logic system. However, it could not do such simple problems as: given "Every person has two hands" and "John is a person", answer the question "How many hands does John have?"

1.3 Resolution

An alternative way to provide deduction with a computer system was clearly required. Researchers turned from ad hoc systems to uniform procedures suggested by mathematical logic. The search for such procedures dates back to Leibniz (1646-1716). The first effective procedure was found by Herbrand in 1930. It was, in fact, implemented on a computer by Gilmore [12]. This early work showed that more efficient techniques were necessary. Davis and Putnam [8] found several ways to improve Herbrand's procedure in 1960. Darlington [7] implemented their system in 1962. A major

breakthrough in theorem proving was the development of the resolution technique by Robinson in 1965 [30]. Resolution has two vital properties. First it is very easily implemented on a computer. Second, it is simple to work with mathematically.

1.4 Languages Based on Formal Logic

Resolution quickly captured the imagination of AI researchers. Cordell Green and Bertram Raphael developed the QA2 [15] system (1968) using resolution and first order logic. Coles [6] wrote an English-to-logic translator which would change subsets of English to the precise form of QA2 input. Early results using this system were very promising. Several very simple problems were solved, but difficulties began to appear. The speed of operation was very low. Small changes in formulation could make it impossible to solve problems. Researchers began to feel that some imperative code was necessary. Green [15] argued that one could conceal imperatives in the declarative formulation. He began to talk of "programming in the predicate calculus for efficiency", and suggested combining imperative and declarative code. He associated a LISP program with selected predicates and functions. The idea, which we will also use, was to execute the LISP program within the resolvent. For example (TIMES 2 2) could evaluate to 4. Using executable predicates,

Green developed QA3. This system could solve the 'Tower of Hanoi' problem, using an executable predicate to check legal moves. Executable predicates were effective, but executable functions were not, for the following reason: After evaluating the executable function, (TIMES 2 2) to 4, it could then be necessary to unify (TIMES 2 X) with 4. Since these cannot be made identical, functional evaluation seemed too difficult. One possible way of unifying (TIMES 2 X) with 4 consists of adding all the equality substitution rules and the axiom $Y = (\text{TIMES } Z (\text{QUOTIENT } Y Z))$; then it would be possible to apply substitution rules enough times to get

$$Z = (\text{TIMES } 2 X) \vee \dots P(X) \dots \vee \dots Q(Z) \dots$$

Resolution with the axiom would give

$$\dots P(\text{QUOTIENT } Y 2) \vee Q(Y) \dots$$

Unification of Y with 4 and finally evaluation of (QUOTIENT 4 2) would then give 2. Unfortunately such an approach leads to an enormous number of resolvents. Since resolution is complete, partial evaluation is useful only when it reduces the number of clauses.

Other researchers questioned the division; "How much imperative vs. how much declarative?" Hewitt [21] developed PLANNER, a language which allowed intermixing fragments of LISP code and predicates within a modified resolution theorem prover. Here one could provide strategy functions which the built-in prover would call to

select clauses. Rulifson et al. developed QA4 [34], a language based on type theory (in order to get an ω -order logic) which allowed clauses to contain programs. These programs, written in a LISP-like notation, would act as a strategy. The difference between QA4 and PLANNER was that PLANNER had a built-in strategy which called user-supplied subroutines, while QA4 had the user program as the entire strategy. Several very impressive programs were written in PLANNER [44]. Unfortunately more and more code took on an imperative flavor. Sussman [41] argued that the built-in strategy of PLANNER was, in fact, the wrong one. In cooperation with McDermott he developed CONNIVER, a language in which users had complete control over the strategy being used. The whole discussion of strategies, often called procedural embedding, is still not ended. One alternative often suggested is that more powerful control structures should be added to the language. A series of languages, QLISP [33], SCHEME [42], etc., were developed to add control flow statements to a logical formalism. We will describe some of the details of these mechanisms in Section 3.4.

It is our feeling that much of this work has been predicated on the beliefs that general purpose uniform procedures had to be very slow, that resolution was developed to its utmost, and that, by their very nature, complete domain-independent strategies could not be made efficient. In our opinion such conclusions are premature. In the next chapter we will describe a generalization of resolution which will be more efficient than resolution alone. We will describe the Dilemma language, which is at about the declarative level of QA3, but is significantly more efficient.

CHAPTER 2

THE THEOREM PROVING SYSTEM

Theorems 1, 2, 4, 5 and 6 in this chapter have been proved elsewhere (Harrison and Rubin [18]); the proofs are repeated here for convenience and completeness.

Resolution Proofs

At this point, we want to consider a difficult resolution problem. This problem, which in this formulation is near the limit which can be proved by current programs, consists in showing that $x \cdot y = (-x) \cdot (-y)$ in a ring. We will use a very simple formulation.

There is one binary predicate equality written as $=$, two binary functions $+$, and \cdot , and a unary function $-$ (minus). We will write the clauses as implications for reasons which will become clear later. The input to the theorem prover is:

Clause	Function
1. $x + z = y + z \rightarrow x = y$	unique sums
2. $x \cdot (y + z) = x \cdot y + x \cdot z$	distributive law
3. $x \cdot y + z \cdot y = (x + z) \cdot y$	
4. $0 = x + (-x)$	additive inverse
5. $0 = -x + x$	
6. $0 \cdot x = 0$	a property of zero
7. $0 = x \cdot 0$	
8. $x = z \wedge z = y \rightarrow y = z$	transitivity of equality
9. $y = x \rightarrow x = y$	symmetric property of equality

- | | | |
|-----|--|--|
| 10. | $x = x$ | reflexive property |
| 11. | $w = y \wedge x \cdot w = z \rightarrow x \cdot y = z$ | } substitution properties
of equality |
| 12. | $w = x \wedge w \cdot y = z \rightarrow x \cdot y = z$ | |
| 13. | $w = x \wedge w + y = z \rightarrow x + y = z$ | |
| 14. | $w = y \wedge x + w = z \rightarrow x + y = z$ | |
| 15. | $a \cdot b \neq (-a) \cdot (-b)$ | negated theorem |

Assuming a resolution system which started with the negated theorem we could get any of a large number of essentially equivalent minimal refutations, each containing 13 resolutions. All of these refutations would involve 3 instances of clause 8, and 1 instance of each of clauses 1,2,3,4,5,6,7,9, 11,12,15. Such refutations would differ only in the order the refutations are done. For example the left distributive law could be used before or after the right.

2.2 Refutation Graphs

We would like to avoid regarding these as different proofs. To that end, we introduce the following definition.

We define a refutation graph G of a set S of unsatisfiable clauses as follows. Let R be a refutation of S . Consider an undirected graph G whose nodes are the clauses of S used in R (if a clause of S occurs more than once in R , there should be one node for each occurrence). Each branch in G is associated with two literals which resolve in R , and connects the clauses containing those literals (if merging or factoring occurs in R , there is a branch in G connecting every clause containing a

literal which merges to become the positive literal to every clause containing a literal which merges to become the negative literal). Figure 1 gives an example of a refutation graph. It is clear that a set S may have many refutation graphs, and that each refutation graph can be derived from one or more resolution refutations.

Performing a resolution on a refutation graph involves replacing the parent clauses by the resolvent, and applying the most general unifier to the other clauses of the graph. The literals of a resolvent will be considered to retain the association they had in the parent clauses. The definition of refutation graph used in this work is similar to but not identical with the resolution graph given in [45].

Figure 2 is a refutation graph of the set of clauses of the input problem. It is easy to see that this particular graph forms a tree, i.e. it has no cycles. Refutation trees are derived from resolution refutations which do not use merging or factoring (in fact from refutations which treat different occurrences of the same literal independently).

We will not consider the refutation graphs which are not trees until Chapter 6.

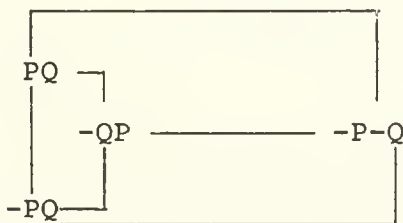


Figure 1

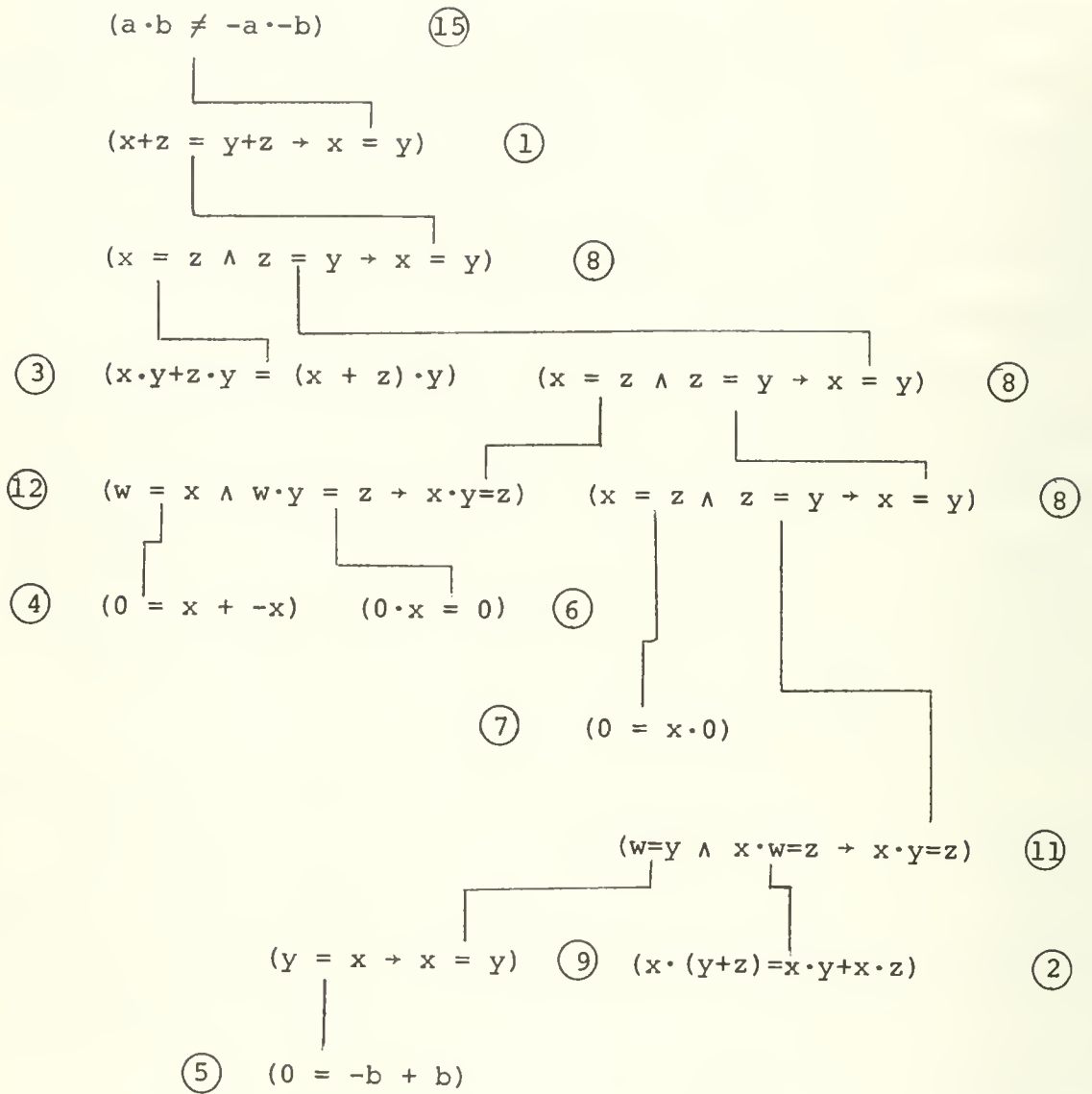


Figure 2. The Refutation Graph of the $x \cdot y = -x \cdot -y$ Example.

If an unsatisfiable set of clauses has a refutation tree then it has a particularly easy sort of refutation.

Before we show this, we need to introduce the following definitions.

A simultaneous unifier (su) of a set of pairs $\langle x_i, y_i \rangle$, $i = 1, \dots, n$, is a substitution λ which satisfies $x_i \lambda = y_i \lambda$, $i = 1, \dots, n$. If there exists an su for a set of pairs, the set is called simultaneously unifiable (which we also signify by su). We use mgu to stand for most general unifier. If the pairs $\langle x_i, y_i \rangle$ are su, then their most general simultaneous unifier (mgsu) is any λ such that for any su μ there exists a σ such that $\lambda \sigma = \mu$.

Lemma 1 (Andrews, 1968). The mgsu of a set of su pairs $\langle x_i, y_i \rangle$ is unique to within alphabetic variations, and is computable as the product $\lambda_1 \lambda_2$, where λ_1 is the mgu of $\langle x_j, y_j \rangle$ and λ_2 is the mgsu of $\langle x_i, y_i \rangle \lambda_1$ for $i = 1, 2, \dots, j-1, j+1, \dots, n$, and j can be chosen arbitrarily.

Proof: This was first proved, in a slightly more general form, in [2]. The proof is by induction on n . If $n = 1$ then the lemma is obvious.

We assume that there is a su called s of $\langle x_i, y_i \rangle$, $1 \leq i \leq n+1$, and a mgsu σ of $\langle x_i, y_i \rangle$, $1 \leq i \leq n$. Let $L = \langle x_{n+1}, y_{n+1} \rangle$.

Since s is an su of the pairs up to n , there is a B such that $s = \sigma b$. Now $Ls = L\sigma b$ is unified, so b unifies $L\sigma$.

Since $L\sigma$ is unifiable there is a mgu τ of $L\sigma$. We assert that $\sigma\tau$ is the mgsu of the pairs up to $n+1$.

- (1) $\sigma\tau$ is an su for $1 \leq i \leq n$, $\langle x_i, y_i \rangle_{\sigma}$ is unified so $\langle x_i, y_i \rangle_{\sigma\tau}$ is also. $L\sigma\tau$ is also unified by the definition of τ .
- (2) $\sigma\tau$ is an mgsu. Suppose γ is an su of the pairs up to $n+1$; then γ is an su of the pairs up to n . So $\gamma = \sigma u$. Since $L\gamma = L\sigma u$, u unifies $L\sigma$, thus $u = \tau c$. Now $\gamma = \sigma u = \sigma(\tau c) = (\sigma\tau)c$.
- (3) Uniqueness. Assume τ, σ are two mgsu; then there exists d, e such that

$$\tau = \sigma d, \quad \sigma = \tau e$$

if P is any well formed formula then $P\tau$ and $P\sigma$ are alphabetic variants, so the unifier is unique to within alphabetic variants.

Now we may prove

Theorem 1 Given a refutation tree

T for a set S of clauses, a merge-free and factor-free resolution refutation of S can be obtained from T by carrying out the resolutions corresponding to the branches of T in any order.

Proof: The refutation tree T contains a set of clauses each of which is in S , and determines the existence of an mgsu of the set of pairs of literals associated with the branches of T . Consider a particular order of the branches

of T, B_1, B_2, \dots, B_n . We can perform the resolutions on the literals associated with the branches in this order. Since an mgsu of the pairs of literals exists, each of these resolutions can be performed. \square

Now we will characterize what class of sets of clauses have a refutation tree.

Theorem 2 The following statements about a set S of clauses are equivalent:

- (i) S has an input proof, using factoring and resolution.
- (ii) S has a unit proof, using factoring and resolution.
- (iii) S plus the factors of S have a refutation tree.

Proof: (i) and (ii) were proved equivalent by Chang [4], so we will just prove the equivalence of (i) and (iii).

First, suppose S has an input proof I , using the clauses $I_0, I_1, I_2, \dots, I_n$ in that order. Consider the refutation graph G of S corresponding to I . Process G as follows:

1. Set $i = n$.
2. If two literals of I_i are associated with the same literal in another clauses, replace I_i in G by the factor of I_i which unifies these literals, making the corresponding modification in the branches of G and go to 2.

3. If a literal L in I_i is associated with two other literals L_1 and L_2 , do the following: Disconnect the branch associated with L and L_1 ; duplicate the graph containing I_i neglecting all branches between I_i and I_j for $j < i$; reconnect this graph by adding a branch associated with L_1 and the copy

of L ; and go to 3.

4. Set $i = i-1$; if i is nonzero, go to 2.

Each of the modifications in steps 2 and 3 does not destroy the fact that G is a refutation graph. Furthermore, this algorithm changes G to a tree, since in step 3 the graph containing I_i is a tree. This can be seen by noting that just before step 4 each copy of I_i is connected to exactly one I_j for some $j < i$.

Now suppose S plus its factors has a refutation tree T . We can carry out the resolutions specified by the tree in any order, so we can construct an input proof by selecting an arbitrary clause of T first, and then performing the resolutions of T with input clauses (or factors of input clauses) in such a way that the literal resolved on is a member of the previous resolvent. \square

Corollary 1 If S has an input (unit) refutation using factoring and resolution, then S plus its factors has an input (unit) resolution refutation without factoring.

Corollary 2 If S has an input refutation using resolution without factoring then S has a refutation tree.

2.3 Horn Clauses

Theorem 2 tells us when a refutation tree exists, but in a rather useless way. We first have to know if a set of clauses has a refutation of a certain form, so we have to give

a proof before we know if there is a refutation tree.

We will now give syntactic conditions which will be sufficient to guarantee the existence of a refutation tree.

If we have a clause in disjunctive normal form

$$\neg L_1 \vee \neg L_2 \vee L_3$$

which has no more than one positive literal, we may write it as an implication.

$$L_1 \wedge L_2 \rightarrow L_3$$

Each such implication is a conjunction of zero or more atoms, followed by an implication sign, followed by either zero or one atom. A clause in this form is called a Horn clause. A set representing a conjunction of Horn clauses is called a Horn set. Henschen and Wos [19] showed:

If S is an unsatisfiable Horn set, then S has an input refutation without factoring.

Theorem 3. If S is an unsatisfiable Horn set, S has a refutation tree.

Proof: From the second corollary to Theorem 2.

Given a set of clauses in Horn form, a possible algorithm for finding refutation trees consists of:

- (1) Form a pool of subtrees initially containing only negative clauses
- (2) Select some subtree in the pool, call it T;
- (3) Remove T from the pool.
- (4) Pick some literal in T which has no associated branch.
- (5) Compute all possible subtrees which link the selected literal of T to positive literals of input clauses.
- (6) If any of these trees has no literal without an associated branch, otherwise add each of these trees to the pool and go to step (2).

It is easy to see that this algorithm is exponential in the number of branches of the final tree. In the next section we will show how to cut down the number of branches and thus reduce the complexity of this algorithm.

2.4 The Revised Unification Problem

In Figure 3, we draw the outline of the $x \cdot y = (-x) \cdot (-y)$ refutation tree.

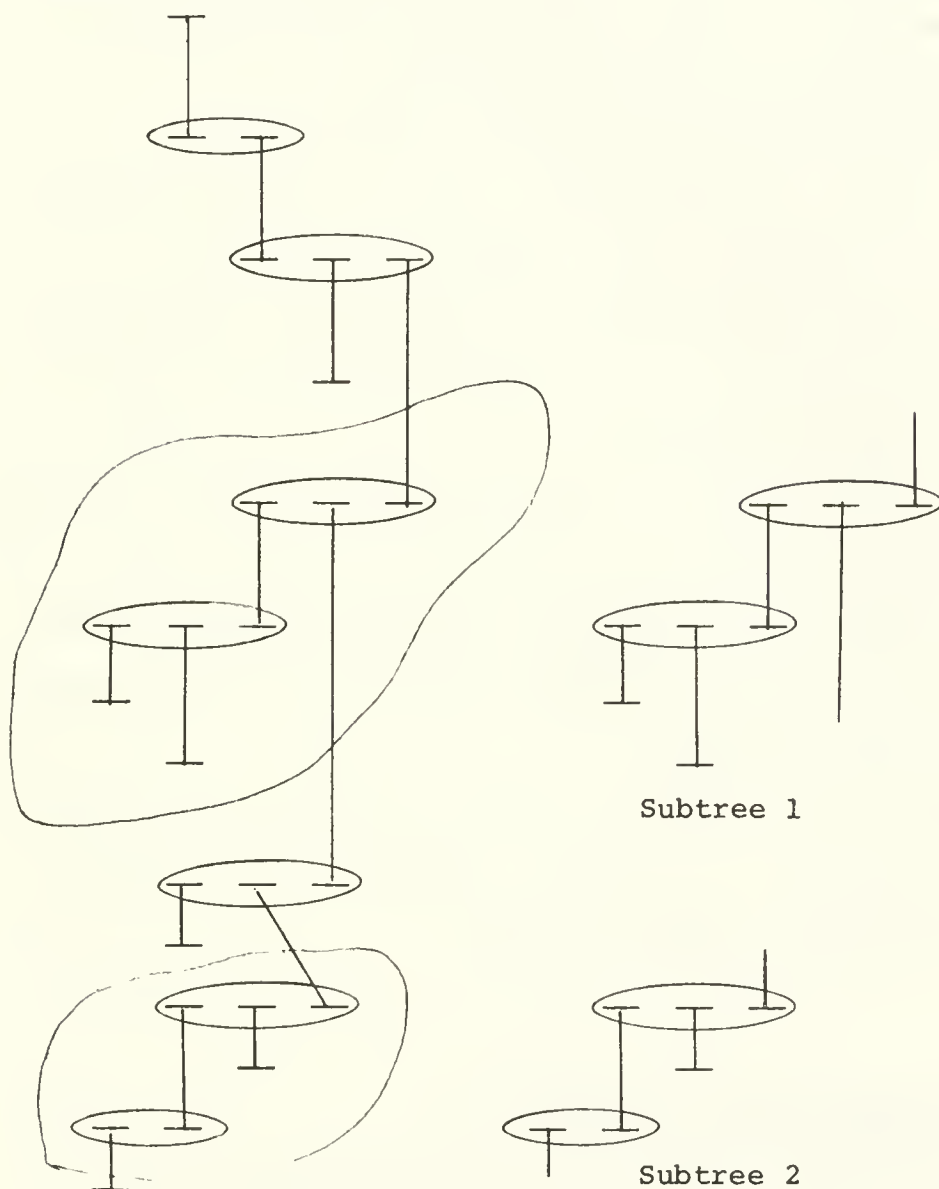


Figure 3. Outline of the Refutation Tree for $x \cdot y = (-x) \cdot (-y)$.

Here, we have selected several subtrees. If we regard the subtrees as units then the resultant tree has fewer branches. We may define a generalization of resolution based on this observation. In what follows, we will always assume that the variables in two clauses are distinct.

Consider two clauses G and H of S of the form $A \cup \{L(x_1, \dots, x_\ell)\}$ and $B \cup \{\bar{M}(y_1, \dots, y_m)\}$. Suppose clause C of the form $D \cup \{\bar{L}(w_1, \dots, w_\ell)\} \cup \{M(z_1, \dots, z_m)\}$ can be deduced from $U \subseteq S$, and there is a most general simultaneous unifier λ of the pairs $\langle x_i, w_i \rangle$, $i = 1, \dots, \ell$, and $\langle y_j, z_j \rangle$, $j = 1, \dots, m$. Then the clause

$$(A\lambda - L(x_1, \dots, x_\ell)\lambda) \cup (B\lambda - \bar{M}(y_1, \dots, y_m)\lambda) \cup (D\lambda - \bar{L}(x_1, \dots, x_\ell)\lambda - M(y_1, \dots, y_m)\lambda)$$

is a binary U-generalized resolvent of G and H. We will refer to clause C as the unifying clause, and to the L and M literals of C as the linking literals.

Furthermore, if clause C' of the form $\{\bar{L}(w_1, \dots, w_\ell)\}$ can be deduced from $U \subseteq S$, and there is an mgsu λ of the pairs $\langle x_i, w_i \rangle$, $i = 1, \dots, \ell$, then the clause

$$A\lambda - \{L(x_1, \dots, x_\ell)\lambda\}$$

is a unary U-generalized resolvent of G.

We will usually shorten U-generalized resolvent to U-g-resolvent, and drop the words "unary" or "binary" when no distinction is being made.

We envisage U-g-resolution being used in the following way. Given a set S of clauses, select a subset U not including the theorem to be proved. Attempt to generate the null clause by repeatedly adding to S the U-g-resolvents of clauses in $S-U$. Thus the clauses in U are used essentially as rules of inference rather than clauses from which inferences can be made. Note that the proofs generated in this way should have fewer steps than standard resolution proofs, and these steps should be larger. The search strategy will involve the selection of (one or) two clauses from $S-U$, and the selection of (one or) two literals belonging to them which can be "linked" by a clause inferred from U .

The question then arises of the completeness and soundness of generalized resolution.

Theorem 4 U-g-resolution is sound.

Proof: Trivial.

Theorem 5 If S is a set of clauses with a refutation tree, then for any satisfiable $U \subset S$ $S-U$ has a U-g-resolution refutation.

Proof: Consider a refutation tree T of S . We will show that the order of performing resolutions in T , which can be chosen arbitrarily, can be chosen so that it consists of a set of subsequences, each of which corresponds to a U-g-resolution step.

Either there is no clause in $U \cap T$, in which case $S-U$ has a standard resolution refutation, or we can find a subgraph t of T satisfying:

- (a) t has at least one node in U
- (b) if a clause K is in $U \cap t$, all clauses connected in T to K are also in t
- (c) t is connected
- (d) all nodes of t not in U are connected in t to only one other node of t .

Thus t is a free tree with internal nodes in U .

Now since U is satisfiable, there must be at least one node of t in $S-U$. If there is exactly one, say K , then it must be possible to carry out all the resolutions in $t - \{K\}$, leaving a single literal to resolve with a literal in K ; in this case the result of carrying out all resolutions in t is a unary U -g-resolvent of K . Otherwise there are at least two nodes of t in $S-U$, say K_1 and K_2 . K_1 and K_2 can be connected by a path P in t , and carrying out the resolutions along P is equivalent to forming a U -g-resolvent of K_1 and K_2 . Note that in carrying out the U -g-resolution, more literals than K_1 and K_2 may be removed from T . However, this merely decomposes T into disconnected subtrees, one of which must be a refutation.

Thus in each case we can reduce T to T' , a tree with strictly less branches than T , by carrying out resolutions corresponding to the formation of some U -g-resolvent R , with T' being a refutation tree for $S \cup \{R\}$.

Thus given any refutation tree T of S we can reduce it by sets of resolutions each corresponding to U -g-resolution steps until the refutation tree contains no more clauses of U . This reduced tree can be refuted by standard resolution steps not involving clauses of U , thus completing a U -g-resolution refutation of $S-U$. \square

Corollary. Since in the proof above we do not require that identical literals be merged, the theorem remains true if we do only partial merging. In particular, if no two literals are ever identified.

In the case of binary U -g-resolution, we may view the generalization as a modification of unification.

Before: we had to find a substitution σ which would make two formulas identical.

Now: We need to find a substitution σ and a subtree T which make the formulas equivalent.

One way in which we may establish equivalence is to use a second, inner, theorem prover. Such an inner theorem prover provides three advantages:

- (1) It needs to do simpler deductions
- (2) The pair of literals selected by the outer theorem prover acts as a guide or higher level plan
- (3) The outer theorem prover reasons in larger steps.

At best, we trade one proof of n steps for two each of about $n/2$ steps. Since the work is exponential in the number of steps, if we happen to be in the best case we get a

2.5 Equality

Silbert [39] advanced the first special equality system. He had four rules of inference.

Rule 2

From $A \cup x = y, B \cup u \neq v$

where A, B only contain equalities or inequalities

y occurs only at the top level in A, σ unifies y

and v, infer $(A \cup B \cup u \neq x)\sigma$

[equality replacement]

Rule 3 From $A \cup F(x_1, \dots, x_n) = y$ where A only contains equalities or inequalities, infer
 $A \cup z_1 \neq x_1 \cup \dots \cup z_n \neq x_n \cup F(z_1, \dots, z_n) = y$
[equality substitution]

Rule 4 Delete literals of the form $x = x$.

The problem in Silbert's plan however where to apply rules 2 and 3. Using them everywhere allows all possible substitutions. His technique was never implemented.

The next proposed equality technique was paramodulation Suggested by Robinson and Wos in 1969 [31], it became the most often used equality method. In summary, it consists of the following.

Given two clauses

$$\begin{array}{c} C \\ x=y \cup B \end{array}$$

where

- (1) t is a term in C
- (2) σ is the mgu of t and x ,

infer the clause

$$C_0 = (C \cup B)\sigma$$

Suppose that there are n occurrences of t , σ in C_0 ; then we conclude the n clauses C_1 through C_n where C_i is the result of replacing the i th occurrence of $t\sigma$ in C_0 with $y\sigma$.

Robinson and Wos showed that "a ring is commutative if $\forall x, x^3 = e$ " could be proved in 49 steps of paramodulation or

136 resolutions. The proof of $x \cdot y = (-x) \cdot (-y)$ in the formulation above takes 11 paramodulation steps or 13 resolutions. Much work has been done with paramodulation, perhaps because it is easy to work with mathematically. Implementations have not shown clearly that it is efficient.

E-resolution, due to Anderson [1], uses both paramodulation and resolution. Two literals are selected; then using the unit equalities, all possible paramodulations of each literal are performed. Then all pairwise ways of unifying the literals are computed. An incomplete version was implemented [27].

Dixon [9] suggested Z-resolution. Basically, it consists of the following: to resolve two clauses $L_1 \vee A$ and $\neg L_2 \vee B$, first compute the sets L_1 -VAR (of equality variants of L_1) and L_2 -VAR (the set of equality variants of L_2). Next compute a set U of unifiers. Each $\sigma \in U$ will unify a member of L_1 -VAR with some member of L_2 -VAR. Finally, the Z-resolvents are computed by

$$(A \cup B)\sigma = \{L_1\sigma, \neg L_2\sigma\} \quad \text{for } \forall \sigma \in U ; .$$

The equality variants of a literal l are obtained as the resolvents of l and clauses in a variant set V . V contains two literal clauses, where no variable occurs more than once in a single literal. Resolutions of elements of V produce tautologies. Z-resolution is a very powerful scheme and is perhaps the best of the equality methods. A particularly attractive alternative to Z-resolution is the following variation of our U-g-resolution technique.

2.6 Equality-Generalized Resolution

Consider two clauses G and H of the form $A \cup \{L(x_1, \dots, x_\ell)\}$ and $B \cup \{\bar{L}(y_1, \dots, y_\ell)\}$. Suppose clauses $D_i \cup \{w_i = z_i\}$ can be deduced from the set U of clauses such that either 1) there is a msgu λ of the pairs $\langle x_i, w_i \rangle$ and $\langle y_i, z_i \rangle$. Or, 2) if in addition L is the equality predicate, there is an msgu λ of the pairs $\langle x_1, w_1 \rangle, \langle z_1, y_2 \rangle, \langle x_2, w_2 \rangle, \langle z_2, y_1 \rangle$. Then the clause

$$(G\lambda - L(x_1, \dots, x_\ell)\lambda) \cup (B\lambda - \bar{L}(x_1, \dots, x_\ell)\lambda) \cup \left(\bigcup_i (D_i\lambda - L(x_1, \dots, x_\ell)\lambda - \bar{L}(y_1 \dots y)\lambda) \right)$$

is a U-equality-generalized resolvent of G and H . In what follows we will shorten U-equality-generalized resolvent to U-e-g-resolvent.

Note that for every U-e-g-resolvent there is a corresponding U-g-resolvent, but the reverse is not true.

In what follows we will refer frequently to the (infinite) set E containing the following equality axioms

$$\{x \neq y, y = x\}$$

$$\{x \neq y, y \neq z, x = z\}$$

and all axioms of the form

$$\{x \neq y, z \neq f(\dots, x, \dots), z = f(\dots, y, \dots)\}$$

$$\{x \neq y, \bar{p}(\dots, x, \dots), p(\dots, y, \dots)\}$$

for all function letters f and predicate letters p .

In practice we will always be concerned with the (finite) subset of E containing only function and predicate letters used in other clauses of a set $S-E$.

Theorem 6 If S is a set of clauses, including $\{x = x\}$, with a refutation tree, then there exists a linear U-e-g-resolvent refutation of $S-U$ for any U satisfying

$$(i) \quad E \cap (S - U) = \emptyset$$

$$(ii) \quad U-E \text{ contains only positive unit equality clauses but not } \{x=x\}.$$

Proof: The proof of this theorem is similar to that of Theorem 5. That is, any refutation tree T with at least one node in U has a subgraph t which is a free tree with internal nodes in U .

We can show that there is at least one pair of (terminal) nodes in $t \cap (S - U)$ which have a U-e-g-resolvent which can be obtained by carrying out only resolutions specified by t . For any internal node of t is in E , so is of one of the following forms:

$$\{x \neq y, y = x\}$$

$$\{x \neq y, y \neq z, x = z\}$$

$$\{z \neq f(\dots, x, \dots), x \neq y, z = f(\dots, y, \dots)\}$$

$$\{\bar{p}(\dots, x, \dots), x \neq y, p(\dots, y, \dots)\}$$

In each case the first literal has the same form as the last literal except for a substitution or, in the case of the

symmetry axiom, reversal of the arguments. This means that starting with any negative literal in t , we can find a sequence of resolutions in which all the resolved literals are identical except for substitutions (or argument reversals, if an equality literal).

Now we can choose this initial negative literal $\bar{L}(x_1, \dots, x_\ell)$ to be in $S - U$. There must be at least one such literal, since not all positive literals in $t \cap U$ can be linked in t to negative literals in $t \cap U$ (this would give a path which would not end in a terminal node, contradicting the condition that t is a tree). The substitution path starting with $\bar{L}(x_1, \dots, x_\ell)$, where L is not equality, must terminate in another clause in $S - U$ containing a linking literal of the form $L(y_1, \dots, y_\ell)$. Furthermore, the internal nodes on this path specify substitutions or reversals which indicate how $\bar{L}(x_1, \dots, x_\ell)$ can be transformed into $L(y_1, \dots, y_\ell)$. These internal nodes, together with some applications of the transitivity axiom if necessary, permit us to deduce from E clauses of the form

$$D_i \cup \{x_i = y_i\}$$

where the D_i are sets of equality literals. Thus the application of the resolutions along such a path corresponds to an E-e-g-resolution step.

If, on the other hand, L is equality, the substitution/reversal sequence of internal nodes starting with $x_1 = x_2$ may end either in a node $\{y_1 = y_2\}$ in $S - U$, or in a positive

equality unit $\{y_1 = y_2\}$ in U . In both cases the set of internal nodes must enable us to define from U clauses of the form

$$D_1 \cup \{w_1 = z_1\}$$

$$D_2 \cup \{w_2 = z_2\}$$

with either $\langle x_1, w_1 \rangle, \langle x_2, w_2 \rangle, \langle y_1, z_1 \rangle, \langle y_2, z_2 \rangle$ simultaneously unifiable, or $\langle x_1, w_1 \rangle, \langle z_1, y_2 \rangle, \langle x_2, w_2 \rangle, \langle z_2, y_1 \rangle$ simultaneously unifiable. In the first case this sequence of resolutions is equivalent to a U-e-g-resolvent with linking literals $x_1 = x_2$ and $y_1 = y_2$. In the second case we can replace $y_1 = y_2$ in t with

$$\{y_1 \neq y_2, y_2 \neq y_2, y_1 = y_2\}$$

$$\{y_1 = y_2\}$$

$$\{y_2 = y_2\}$$

the last literal being an instance of $x = x$, which is in $S-U$. Thus the second case can be reduced to the first, so also corresponds to an U-e-g-resolution step. Thus given any refutation tree T of S , we can reduce that tree by applying sets of resolutions corresponding to U-e-g-resolution steps until the tree contains no more clauses of U . This reduced tree can be refuted by standard resolution steps not involving clauses of U , thus giving an U-e-g-resolution refutation. \square

Corollary. As in Theorem 5, Theorem 6 remains true whether merging of literals is done or not.

In effect, Theorem 6 says that if in U-g-resolution U is constrained to be E, then we need only consider unifying clauses which have linking literals with the same predicate letter, and further, such unifying clauses have as their essential components other clauses which establish the equality of the arguments of the linking literals.

Theorem 6 thus gives us a generalization of the standard unification procedure. The standard procedure says that literals can be unified only if there are substitutions for variables which make the literals identical; the generalized procedure is to permit substitutions which enable the arguments of the literals to be proven equal. Theorem 6 shows that in the case of sets of clauses with a tree refutation the use of the equality axioms can be restricted to the problem of proving the arguments equal, and need not appear as clauses to be resolved upon.

Theorem 6 cannot be generalized to arbitrary sets of clauses; the E-unsatisfiable set of clauses:

$$\{\bar{P}(a), \bar{P}(b)\}$$

$$\{P(a), P(b)\}$$

$$a = b$$

cannot be proved unsatisfiable by the E-e-g-resolution procedure since the necessary factoring cannot be done. We conjecture, however, that if factoring and merging is generalized to use equivalence in an analogous fashion, then many of the completeness results of standard resolution procedures can be carried over.

Returning to the example, no proof of $x \cdot y = -x \cdot -y$ in the formulation given in Section 2.1, has been reported in the literature, though Wos et al. [31] proved it in a hybrid formulation (using a three-argument product predicate and also the equality predicate). Our formulation, being restricted to Horn clauses, must have an input proof starting with the negated theorem. The following is a standard input resolution proof of minimum depth (13 resolvents): Numbers to the right correspond to Section 2.1.

- (R1) $a \cdot b + x \neq -a \cdot -b + x$ from (15) and (7)
- (R2) $a \cdot b + x \neq y, y \neq -a \cdot -b + x$ (R1) and (8)
- (R3) $(a+b) \cdot b \neq -a \cdot -b + x \cdot b$ (R2) and (6)
- (R4) $(a+b) \cdot b \neq y, y \neq -a \cdot -b + x \cdot b$ (R3) and (8)
- (R5) $w \neq a + x, w \cdot b \neq y, y \neq -a \cdot -b + x \cdot b$ (R4) and (11)
- (R6) $0 \cdot b \neq y, y \neq -a \cdot -b + -a \cdot b$ (R5) and (3)
- (R7) $0 \neq -a \cdot -b + -a \cdot b$ (R6) and (2)
- (R8) $0 \neq x, x \neq -a \cdot -b + -a \cdot b$ (R7) and (8)
- (R9) $x \cdot 0 \neq -a \cdot -b + -a \cdot b$ (R8) and (1)
- (R10) $w = 0, x \cdot w \neq -a \cdot -b + -a \cdot b$ (R9) and (12)
- (R11) $0 \neq w, x \cdot w \neq -a \cdot -b + -a \cdot b$ (R10) and (10)
- (R12) $x \cdot (-y+y) \neq -a \cdot -b + -a \cdot b$ (R11) and (4)
- (R13) \square (R12) and (5)

The refutation tree corresponding to this proof is given in Figure 2. Note that axioms (9), (13), and (14) are not used, but that (10) would have been necessary in several other cases had other axioms (such as (1), (2), (5), and (6)) been specified the other way around.

A paramodulation proof would permit the elimination of the steps involving the substitution axioms (11) and (12). That is, (R6) could be obtained directly from (R4), and (R11) could be obtained directly from (R9). This gives an 11-step proof.

A proof using E-e-g-resolution can be much shorter, since all resolvents involving clauses in E are omitted. That is, only resolvents (R1), (R3), (R6), (R7), (R9), (R12) and (R13) appear explicitly, giving a 7-step proof. This proof arises in a number of ways: for example, an E-e-g-resolvent clause could be found from (R1) and (6), using one instances of (8); or (R1) and (2), using two instances of (8) and one of (11); or (R1) and (1) using three instances of (8); or (R1) and (5) using three instances of (8) and one of (12). In practice it would not be likely that the longer unifying clauses would be found; it would be more likely that (R1) would resolve with (5) using a simple substitution for the right-hand sides, and an instance of (8)

$$a \cdot b + -a \cdot z' \neq a, z \neq x' \cdot (y' + z'), a \cdot b + -a \cdot z' = x' \cdot (y' + z')$$

for the left-hand sides.

The S-e-g-resolution proof is even shorter than this, consisting of only 2 steps, (R1) and (R12). That is, (R1)

$$a \cdot b + x \neq -a \cdot -b + x$$

can be resolved directly with (5)

$$x' \cdot (y' + z') = x' \cdot y' + x' \cdot z'$$

The right-hand sides are unifiable immediately, using the substitutions $x' \leftarrow -a$, $y' \leftarrow -b$, $x \leftarrow -a \cdot z'$. Generalized unification of the left-hand sides requires that a substitution be found such that $(a \cdot b + -a \cdot z')\lambda$ can be proved equal to $(-a \cdot (-b + z'))\lambda$. This can be done, following the refutation tree precisely, as follows:

$$\begin{array}{ll}
 a \cdot b + -a \cdot b = (a + -a) \cdot b & \text{from (6)} \\
 a \cdot b + -a \cdot b = 0 \cdot b & \text{from (3) and (11)} \\
 a \cdot b + -a \cdot b = 0 & \text{from (2) and (8)} \\
 a \cdot b + -a \cdot b = -a \cdot 0 & \text{from (1) and (8)} \\
 a \cdot b + -a \cdot b = -a \cdot (-b + b) & \text{from (4), (10) and (12)}
 \end{array}$$

With the substitution $z' \leftarrow b$, this gives the null clause as the S-e-g-resolvent of (R1) and (5).

2.7 Coercion

To turn Theorem 6 into a useful procedure, we need a strategy for finding the proofs of equality.

Suppose we have two terms t_1 and t_2 which we would like to prove equal. Also suppose U contains a clause

$$a = b \leftarrow D, \text{ where } D \text{ is a conjunction.}$$

We may break up the proof of equality into two subproblems. First to show that $t_1 = a$ and second, to show that $b = t_2$. We must decide which clauses of U to use on a given problem. We choose to use the following criteria.

Before starting to prove that $t_1 = a$ we impose the restriction that either

- (1) t_1 or a is a variable or
- (2) t_1 and a are the same constant or
- (3) t_1 and a both start with the same function letter.

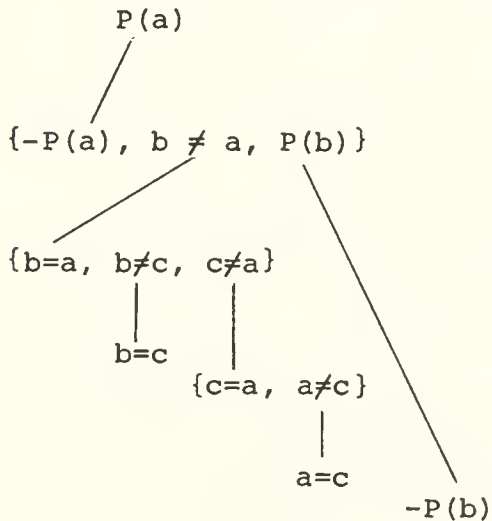
We then use the same criteria on t_2 and b .

Finally we place transitivity in both S and U .

The difference between coercion and U-e-g resolution may be seen in the following example.

<u>S - U</u>	<u>U</u>
$P(a)$	$a = c$
$\neg P(b)$	$b = c$
$x = x$	E

In U-e-g resolution we unify $P(a)$ and $\neg P(b)$ in the following tree.



Unification of $P(a)$ and $\neg P(b)$ produces the subproblem of proving $b = a$, which can be done within U . Thus U-e-g-resolution gives a one-step proof.

In coercion the transitivity axiom is in S rather than in U, so a coercion proof takes four steps. From an attempt to unify $P(a)$ and $\neg P(b)$, again we get the subproblem $b=a$. Now however we cannot apply transitivity inside unification, so the resolvent is

$$\{b \neq a\}$$

In the second step we resolve $b \neq a$ with transitivity to get

$$\{b \neq z, z \neq a\}$$

Two unary resolutions remove these literals. As this example shows, coercion consists in stopping the inner theorem prover just before applying transitivity. Transitivity is then applied by the outer theorem prover.

Finally unary resolutions complete the proof.

The coercion algorithm we use is defined as follows:

An ordered pair $\langle \sigma, A \rangle$, where σ is a set of substitutions and A is a set of literals, is called a solution of the coercion-unification of $L(x_1, \dots, x_n)$ and $\neg L(y_1, \dots, y_n)$ if each of the following conditions holds: (1) a subtree t may be constructed inside of unification; (2) the literals $\neg L(x_1, \dots, x_n)\sigma$ and $L(y_1, \dots, y_n)\sigma$ are in t and have no associated branch; (3) A is the set made up of all the other literals of t without associated branches. The depth of a solution is the number of times that transitivity occurs in the associated subtree.

The following algorithms will generate all solutions of the coercion unification of $L(x_1, \dots, x_n)$ and $L(y_1, \dots, y_n)$ of depth d or less.

To coercion-unify $L(x_1, \dots, x_n)$ and $L(y_1, \dots, y_n)$ to depth d ,

(A) Solve the subproblem

$x_1=y_1, x_2=y_2, \dots, x_n=y_n$ to depth d as follows:

- (1) Let S = set of solutions of the subproblems $x_i = y_i$ to depth d as shown in (B) below
- (2) If L is equality add to S the solutions of $x_1=y_2, x_2=y_1$, as shown in (B) below.
- (3) Return S .

(B) To solve a subproblem

$x_1=y_1, x_2=y_2, \dots, x_n=y_n$

- (1) let S_1 = set of solutions of $x_1=y_1$ to depth d as found by (C) below
- (2) let S_i = set of solutions of $(x_i=y_i)\sigma$ as found by (C) below where $\langle \sigma, A \rangle \in S_{i-1}$ for $i = 2, \dots, n$.
- (3) Return S_n .

(C) To solve a subproblem

$x_i=y_i$ to depth d ,

- (1) let $S = \{ \langle I, x_i=y_i \rangle \}$ where I is the identity substitution
- (2) if x_i and y_i are not both functions, let σ = most general unifier of x_i and y_i computed in the standard way; if σ is defined add $\langle \sigma, \emptyset \rangle$ to S .
- (3) If $d = 0$, go to (6).
- (4) For each unit equality problem $\alpha = \beta$ form the two subproblems $x_i=\alpha, y_i=\beta$ with depth $d-1$; $x_i=\beta, y_i=\alpha$ with depth $d-1$.
- (5) If the restrictions to Section 2.7 are met, add all solutions to these subproblems, found by (B) above to S .

- (6) If x_i and y_i are functions starting with the same symbol add to S all solutions of coercion-unifying x_i and y_i to depth d .
- (7) Return S .

For the unary case, we have:

To coercion-unify $x \neq y$
 apply algorithm (C) above.

Finally, we prove below that we get the same deductions from either coercion or U-e-g-resolution. The advantage of coercion is that the outer theorem prover gets to see intermediate results and on that basis possibly suspend the computation of a unifier.

Theorem 7 If S is a set of clauses, including $\{x \neq y, y \neq z, x = z\}$, with a refutation tree, then there exists a linear coercion refutation of $S-U$ for any U satisfying

- (i) $E \cup (S - U) = \{x \neq y \vee y \neq z \vee x = z\}$
- (ii) $U-E$ contains only positive unit equalities but not $\{x=x\}$.

Proof: We will show that any U-e-g resolvent is produced by a series of coercion resolvents which do not involve any uses of transitivity inside of unification. We do this by induction on n , the number of uses of transitivity in the U-e-g resolution.

If $n = 0$, the U-e-g resolution and the coercion resolution are the same.

Assume the induction hypothesis is true for U-e-g resolutions with n or less transitivity applications. Consider a U-e-g

resolution with $n+1$ transitivity steps. We select one of these $n+1$ transitivity applications, say:

$$\{a \neq c, c \neq b, a = b\}$$

The $a = b$ must be linked to an $a \neq b$. If this $a \neq b$ is in a clause in $S-U$ then we may resolve $a \neq b$ and transitivity directly. Otherwise this $a \neq b$ is in U , and therefore in E , since $U-E$ contains only positive unit equalities. There are two cases: either it is on a chain of the type described in Theorem 6, or it is not. In the first case it can be removed by a $U-e-g$ -resolution, and in the second must be produced by some $U-e-g$ resolution as one of the resultant literals. In both cases this $U-e-g$ -resolution will involve no more than U applications of transitivity, and so can be done by coercion.

Finally the two literals $a \neq c$, and $c \neq b$ may each be removed by a unary coercion. Such a deduction is allowed since the subtrees linked to these literals in the $U-e-g$ resolution each contains no more than n applications of transitivity.

We have shown that no applications of transitivity are required inside unification; since using transitivity inside of unification can only increase the number of solutions, we may apply it selectively without removing completeness. Therefore in part (C) step (5) any set of restrictions may be used.

2.8 Comparison of Coercion and Resolution

The results reported below were obtained from a straightforward Horn clause resolution program which was modified to implement coercion. A switch could be set in this algorithm so that it would revert to the standard unification algorithm for comparison purposes.

Horn clause resolution was chosen for its simplicity. We use input resolution, with the literals in a clause maintained in the original order, and resolution permitted only on the first literal. We prefer input resolution, since at any point in an input resolution procedure, there are a number of resolvents which can be worked on (the "pool") and the order in which they are chosen does not affect the possible resolutions which can be done; this would appear to be a significant advantage when trying to compare procedures which are highly dependent on heuristics.

Only three heuristics were actually used. Resolvents with complexity (in effect the number of symbols in the expression) greater than some cut-off point were discarded; resolvents were selected in order of increasing complexity and increasing depth if complexity was equal; coercion was restricted to a maximum depth of three. Resolvents which were alphabetic variants of previously encountered clauses were also discarded.

Example 1 is the ring theory problem discussed in Section 2 above. The generalized unification procedure found the proof discussed, using axioms (1) through (8) only. The results are summarized in Table I. The standard resolution procedure ran out of time after having generated 30% more resolvents but only getting half of the proof.

Example 2 is a somewhat simpler ring theory problem. That is, from the axioms

$$(1) \quad x = 0 + x$$

$$(2) \quad x \cdot (y+z) = x \cdot y + x \cdot z$$

$$(3) \quad x + z = y + z \rightarrow x = y$$

together with appropriate equality axioms, to prove

$$(4) \quad x \cdot 0 = 0$$

In standard resolution, this proof could be obtained in the following 11 steps:

(R0)	$a \cdot 0 \neq 0$	negated theorem
(R1)	$a \cdot 0 + z \neq 0 + z$	(R0) and (3)
(R2)	$0 + z \neq a \cdot 0 + z$	(R1) and symmetry
(R3)	$x \neq y, 0 + y \neq a \cdot 0 + x$	(R2) and substitutivity
(R4)	$0 + (x \cdot y + x \cdot z) \neq a \cdot 0 + x(y+z)$	(R3) and (2)
(R5)	$a \cdot 0 + x \cdot (y+z) \neq 0 + (x \cdot y + x \cdot z)$	(R4) and symmetry
(R6)	$v \neq x \cdot (y+z), a \cdot 0 + v \neq 0 + (x \cdot y + x \cdot z)$	(R5) and substitutivity
(R7)	$x \cdot (y+z) \neq v, a \cdot 0 + v \neq 0 + (x \cdot y + x \cdot z)$	(R6) and symmetry
(R8)	$w \neq y+z, x \cdot w \neq v,$ $a \cdot 0 + v \neq 0 + (x \cdot y + x \cdot z)$	(R7) and substitutivity

(R9) $x \cdot w \neq v, a \cdot 0 + v \neq 0 + (x \cdot 0 + x \cdot w)$ (R8) and (1)
 (R10) $a \cdot 0 + x \cdot w \neq 0 + (x \cdot 0 + x \cdot w)$ (R9) and
 (R11) \square (R10) and (1)

Our standard resolution program actually ran out of time after generating 754 clauses, getting (R4) but not (R5). There were 429 clauses left in the pool at this point. Note that this proof is made longer than it might be by the use of the particular forms of the axioms, necessitating several applications of the symmetry axiom.

The generalized unification proof took just the following two steps:

(R0) $a \cdot 0 \neq 0$
 (R1) $a \cdot 0 + x \neq 0 + x$ (R0) and (3)
 (R2) \square (R1) and (1) and (4)

This proof appeared as the 17th clause generated, at which time there were 4 clauses in the pool. The step from (R1) to (R2) is by no means obvious, and proceeds as follows: First (R1) is resolved against (1), with the unification subproblem

$$a \cdot 0 + x = x$$

which is unified with (2) giving the subproblem

$$a \cdot 0 + x \cdot (0 + z) = x \cdot 0 + x \cdot z$$

which generates the subproblem

$$0 + z = z$$

which unifies with (1) reversed. The results are summarized in Table II.

Coercion		Resolvent	Standard Resolution	
Resolvent Number	Pool Size		Resolvent Number	Pool Size
2	0	(R1)	2	0
13	4	(R2)	40	24
21	3	(R3)	156	79
65	14	(R4)	452	214
128	21	(R7)	>779 *	351 *
136	26	(R8)		
370	30	(R9)		
597	46	□		
80 resolvents accepted			* out of time 130 resolvents accepted	
236 seconds			>512 seconds	

Table I. Comparison of Proofs of $a \cdot b = -a \cdot -b$

Coercion		Resolvent	Standard Resolution	
Resolvent Number	Pool Size		Resolvent Number	Pool Size
1	0	(R1)	4	15
not needed	}	(R2)	35	20
		(R3)	57	35
		(R4)	338	208
		(R5)	>754 *	429 *
		...		
17	4	□		
12.7 seconds			* out of time	
			>512	

Table II. Comparison of Proofs of $x \cdot 0 = 0$

CHAPTER 3

THE DILEMMA PROGRAMMING LANGUAGE

While it would be possible to base a computer language solely on the coercion mechanism such a system would then be deficient in several aspects. First, since it would not have a procedural component, though we could state any fact or concept, it would be hard to tell the theorem prover exactly when to use these facts or concepts. We believe that such additional information is often best given in imperative form.

A second deficiency occurs when there are many extraneous ground clauses. If, for example, we wished to have the entire multiplication table of the hardware described by a set of clauses, the storage and retrieval problems would be terribly complex. In any given run only a few of these would be relevant. A far simpler technique would be to generate ground clauses as needed.

The third deficiency is the duplication that arises when there are clauses which are equivalent. For example in a geometry program we could have a clause containing the line AB and a second clause containing the line BA. Ideally we would like to represent clauses in a canonical form deleting duplicates.

The final deficiency that we wish to consider is that duplicate computations are often necessary. If in the course of proving two lines equal we prove that a pair of triangles is congruent, we would like to record that fact. Proofs of congruence should only be done once. Furthermore, once we know that two triangles are congruent, there are

a number of immediate consequences; for example, corresponding angles are equal. Often it is computationally advantageous to draw these consequences immediately.

In this chapter we will describe the pure language and a way to remove some of these deficiencies by introducing imperatives. We next will give the syntax and semantics of Dilemma using coercion and imperatives. Chapters 4 and 5 discuss a sample program which proves geometry theorems. Finally, Appendix I contains an interpreter of Dilemma written in BALM.

3.1 Horn Clauses as Procedures

Logical statements of the form:

$$\begin{array}{c} \neg P_1 \vee \neg P_2 \vee \dots \vee \neg P_n \vee Q \\ Q \\ \neg P_1 \vee \neg P_2 \vee \dots \vee \neg P_n \end{array}$$

are said to be in Horn form. Each such statement has at most one positive literal. Many useful structures may be described by a collection of Horn statements. The following result due to Horn [22] characterizes such structures. We first define the notion of "closed under direct product". Suppose S is a set of statements, with m_1 and m_2 two models for S . If $m_1 \times m_2$ is also a model of S we say S is closed under direct product. Horn showed that if a set of clauses was closed under direct product then there exists an equivalent set of clauses which is in Horn form. For example since we know that if G_1 and G_2 are two groups, then $G_1 \times G_2$ is also a group, there is a set of logical statements in Horn clause form which will describe groups.

Statements of the form $\neg P_1 \vee \neg P_2 \vee \dots \vee \neg P_n \vee Q$ may be viewed as implications $P_1 \wedge P_2 \wedge \dots \wedge P_n \rightarrow Q$ and as procedures with name Q and body P_1 through P_n . We interpret such a procedure to mean: P_1 through P_n are sufficient conditions to establish Q .

Statements of the form Q may also be viewed as implications $\rightarrow Q$ or as procedures with name Q , and null body. We interpret such a procedure to mean Q is an immediate result.

Finally statements of the form $\neg(P_1 \vee \dots \vee P_n)$ are also implications

$$P_1 \wedge P_2 \wedge \dots \wedge P_n \rightarrow \text{contradiction}$$

We interpret this as initial goals, i.e. report success if it is possible to achieve all of P_1 through P_n .

When there are arguments involved we coerce them together. For example:

Given the procedures:

- (1) $P(b,a) \rightarrow \text{contradiction}$
- (2) $Q(x,a) \rightarrow P(a,x)$

and the unit equality $a = b$ we execute the body of (1) by first proving that $b = a$ inside of unification and then binding x to a . The body of (2) becomes $Q(a,a)$.

The term coercion is based on the coercions of ALGOL 68.

We note that bindings in a call affect the caller as well as the called program. We could, for example, have the assignment statement:

$\rightarrow \text{SETQ}(x,x)$

which could be called by either $\text{SETQ}(x,z)$ or $\text{SETQ}(z,x)$.

The basic execution semantics of our language consists of:

- (1) let N = set of negative clauses
- (2) pick some c in N ; remove it from N
- (3) pick some literal l in c
- (4) using coercion, compute the set S of generalized resolvents of subroutines with C , resolving on l and the name of the subroutine
- (5) if $\square \in S$, stop; otherwise, go to (2).

As we have said, this form of language would be very inefficient. The next two sections will describe additional features which improve the performance.

Since the basis of the language will be a two level theorem prover applied to Horn clauses, we have chosen to call the language Dilemma. Before we give a formal statement of Dilemma, we will give an example.

3.2 An Example (The Jekyll and Hyde Problem)

Suppose we wish to tell the machine of a kindly old doctor named Jekyll. Jekyll is the name of an individual or a constant in our logical universe. To tell DILEMMA about him we write:

```
CONSTANT JEKYLL END;  
FACT1 ASSERT DOCTOR(JEKYLL);  
FACT2 ASSERT KINDLY(JEKYLL);  
F3 ASSERT OLD(JEKYLL);  
F4 ASSERT HUMAN(JEKYLL);
```

(we have underlined keywords in these examples; the underlining is not part of the actual computer input.) Each of the statements generates a positive unit clause. Now we tell DILEMMA the nature of humans.

```
F5 THEOREM FALLIBLE(X) FROM HUMAN(X);
```

F5 is an implication: for all x, if x is human then x is fallible. To ask the question "is anyone fallible?" we may write:

```
PROVE (FALLIBLE(X));
```

PROVE statements generate negative clauses. In this case NOT FALLIBLE(X). This clause unifies with F5 to give the resolvent -- NOT HUMAN(X). In turn this resolves with F4 to give the resolvent \square . Thus we have proved a fallible individual exists, namely Jekyll.

A slightly more complex question is "is there a fallible doctor?".

PROVE(DOCTOR (X) \wedge FALLIBLE(X));

The literal NOT DOCTOR(X) unifies with FACT1 to give the resolvent -- NOT FALLIBLE (JEKYLL). This generates NOT HUMAN(JEKYLL) via F5. As above a resolution with F4 produces the null clause. If there were a few other doctors in our universe of discourse, say Strangelove and Schweitzer we could add the inputs:

CONSTANT STRANGELOVE, SCHWEITZER END;

F6 ASSERT DOCTOR(STRANGELOVE);

F7 ASSERT DOCTOR(SCHWEITZER);

Asking the same question:

PROVE(DOCTOR(X) \wedge FALLIBLE(X));

gives the following resolvents:

NOT FALLIBLE(STRANGELOVE)	from F6;
NOT FALLIBLE(SCHWEITZER)	from F7;
NOT FALLIBLE(JEKYLL)	from FACT1.

Each of these is executed in parallel with F5 to give:

NOT HUMAN(STRANGELOVE)
NOT HUMAN(SCHWEITZER)
NOT HUMAN(JEKYLL)

The last of these yields the null clause via F4. The X in the PROVE statement is assigned the value JEKYLL since that is the only solution which produces the null clause. The other assignments of X are "undone" as those parallel paths of execution terminate unsuccessfully.

Suppose we add still another individual to this system, the diabolical Mr. Hyde.

CONSTANT HYDE END;
F8 ASSERT DIABOLIC(HYDE);

Up to this point DILEMMA has been similar to older AI languages. Now we come to a basic difference. We will say that Jekyll and Hyde are the same. One way to do this would be to write:

S1 ASSERT SAME(HYDE,JEKYLL);

Of course we must define the effects of SAME.

S2 THEOREM DOCTOR(X) FROM DOCTOR(Y) \wedge SAME(X,Y);
S3 THEOREM KINDLY(X) FROM KINDLY(Y) \wedge SAME(X,Y);
S4 THEOREM OLD(X) FROM OLD(Y) \wedge SAME(X,Y);
S5 THEOREM HUMAN(X) FROM HUMAN(Y) \wedge SAME(X,Y);
S6 THEOREM FALLIBLE(X) FROM FALLIBLE(Y) \wedge SAME(X,Y);
S7 THEOREM SAME(X,Y) FROM SAME(Y,X);
S8 THEOREM SAME(X,Y) FROM SAME(X,Z) \wedge SAME(Z,Y);
S9 ASSERT SAME(X,X);
SA THEOREM DIABOLIC(X) FROM DIABOLIC(Y) \wedge SAME(X,Y);

If we ask "is there a diabolic doctor?"

PROVE (DOCTOR(X) \wedge DIABOLIC(X));

the resolvents are:

1. NOT(DOCTOR(Y) \wedge SAME(X,Y) \wedge DIABOLIC(X))
2. NOT(DIABOLIC(JEKYLL))
3. NOT(DIABOLIC(STRANGELOVE))
4. NOT(DIABOLIC(SCHWEITZER))

In stage two the resolvents are:

- 1, 5. NOT(DOCTOR(Z), SAME(Y,Z), SAME(X,Y), DIABOLIC(X))
6. NOT(SAME(X,JEKYLL), DIABOLIC(X))
7. NOT(SAME(X,STRANGELOVE), DIABOLIC(X))

8. NOT(SAME(X,SCHWEITZER), DIABOLIC(X))
 2, 9. NOT(DIABOLIC(Y), SAME(JEKYLL,Y))
 3,10. NOT(DIABOLIC(Y), SAME(STRANGELOVE,Y))
 4,11. NOT(DIABOLIC(Y), SAME(SCHWEITZER,Y))

In stage three the resolvents include:

NOT(DIABOLIC(HYDE))
 NOT(SAME(JEKYLL,X), DIABOLIC(X))
 NOT(SAME(X,Z), SAME(Z,JEKYLL), DIABOLIC(X))
 NOT(DIABOLIC(JEKYLL))

Finally in stage 4 we produce the null clause.

Another way available in DILEMMA to describe the intimate relationship between Jekyll and Hyde is the coercion

H1 COER HYDE,JEKYLL;

H1 tells DILEMMA that the constant HYDE may be "COERced" into the constant JEKYLL when necessary for unification.

In standard unification we seek to find a substitution which will make two literals identical. In DILEMMA we require only that the literals become equivalent. Coercions are used to specify facts to an inner theorem prover. This prover, invoked during unification, attempts to prove the equivalence.

So, using H1 rather than the S's we have:

PROVE(DOCTOR(X) \wedge DIABOLIC(X));

1. NOT(DIABOLIC(JEKYLL))
2. NOT(DIABOLIC(SCHWEITZER))
3. NOT(DIABOLIC(STRANGELOVE))

In stage 2 we get the null clause by resolving 1 with F8 using H1.

The Dilemma system has been developed as an extension of the BALM4 programming language [16]. All of the BALM4 features are available to the Dilemma programmer. Several additional data types and structures are provided. We will assume that readers are familiar with the BALM4 language and will freely make use of the concepts and terminology defined in [17].

We will use the following notation:

<e1>,<e2>	will be two expressions
<item>	will be any valid BALM4 item
<lit>	will be a literal
[x]*	will be 0 or more occurrences of x
<pos lit>	will be a literal which is interpreted to be positive.
<name>	will be any BALM identifier
<proc>	will be a BALM identifier

3.3 The Syntax of a Basic Dilemma Program

3.3.0 Constants

Constants are defined by the CONSTANT statement in the form:

```
CONSTANT item,item,...,item END;
```

The items may be any expression. Usually, however, they are numbers or BALM4 identifiers. One CONSTANT statement adds to

the next. So if we use:

```
CONSTANT A,B,C,D END;
```

```
CONSTANT 1,2,RTANG,3,4 END;
```

we would have nine constants.

In input form constants are just the items from CONSTANT statements. However, internally (and in output) an extra set of parentheses are added to turn the constant into a function of zero arguments, so the input constant RTANG becomes (RTANG) internally.

3.3.1 Expressions and Literals

Expressions and literals occur in two forms in Dilemma. The first of these is called input form. Here, Skolem functions follow the rules for BALM4 expressions; e.g.

```
F(X),    G(X,Y),    S(X, F(Y)) .
```

Any BALM4 identifier (which has not been in a CONSTANT statement) inside of an expression is a variable and is local to the Dilemma statement containing that expression. Function and predicate symbols are just BALM identifiers. In the same way literals (in input form) follow the rules for BALM4 expressions; e.g., P(F(X)), P(X), Q(F(X), Y). Each time a predicate or function is used it must have the same number of arguments.

The other form called internal form is more LISP-like: all commas are removed, the parentheses are shifted to

include the predicate or function symbol, and variables are changed to numbers; e.g.

$P(F(X), X, F(X))$ is changed to $(P (F1) 1 (F 1))$

Internal form is used for output.

3.3.2 Unit Positive Clauses

The simplest Dilemma statement is the assertion:

`<name> ASSERT <literal>;`

For example:

`L1 ASSERT EQLINE (L(X,Y), L(Y,X));`

Assertions are positive unit clauses; they need not be ground.

3.3.3 Implications

In Dilemma an implication may be written:

`<name> THEOREM <pos literal> FROM <literal> [\wedge <literal>]*;`

The idea is that the conjunction of literals is sufficient to prove the <pos literal>. An example:

`DESCARTES THEOREM EXISTS(X) FROM THINKS(X);`

3.3.4 Negative Clauses

A negative clause is always the last input. It may be specified in one of the following ways:

- (1) PROVE(<lit>); The lit is a unit negative clause
- (2) PROVE(<lit>[\wedge <LIT>]^{*}); There is one negative clause.
Each literal must be proved.
- (3) PROVE(LIST(<lit>[\wedge <lit>]^{*}, ..., <lit>[\wedge <lit>]^{*}));
Prove any one of the conjunctions.

The third form is an "or of ands". The clauses are ordered. It is more efficient to enter clauses in increasing complexity.

The name of an assertion or implication is required. It is used to label output and internally to reference the clause. (Internally, the corresponding clause is actually assigned as the value of the identifier used to name it.)

3.3.5 Coercions

Coercions are used to state that two expressions are equal (or equivalent). If $\langle e1 \rangle$ and $\langle e2 \rangle$ are predicates then we will talk about their equivalence. Otherwise these are expressions and we talk of their equality. In either case we write $\langle e1 \rangle = \langle e2 \rangle$. To say that $\langle e1 \rangle = \langle e2 \rangle$, we write:

$\langle name \rangle \text{ COER } \langle e1 \rangle, \langle e2 \rangle;$

Often we may find a condition under which $\langle e1 \rangle = \langle e2 \rangle$. The general form is

$name \text{ COER } \langle e1 \rangle, \langle e2 \rangle, \langle lit \rangle [\wedge \langle lit \rangle]^*;$

Suppose $H(X)$ implies $F(X) = G(X)$. Then we may write:

$HFG \text{ COER } F(X), G(X), H(X);$

If two ground items are not equal this may be specified by

$\text{DIFFERENT}(\langle item1 \rangle, \langle item2 \rangle)$

3.3.6 CPROCS

In order to have a useful language we need to be able to encode lots of tricks and heuristics. Dilemma provides a number of methods which let the user alter the normal flow of control.

Normally a coercion is used whenever applicable. However, we frequently know the restrictions on when to apply a particular coercion. Suppose we have a coercion $C \text{ COER } X, Y$ and a BALM procedure P ; the statement $C \text{ CPROC } P$ is of the

general form

```
<cname> CPROC <proc>;
```

which associates the procedure and the coercion.

We call P the CPROC of the coercion.

A CPROC is called just before the associated coercion is applied. Each CPROC has three arguments:

1. item from negative clause to be unified
2. item from mixed or positive clause to be unified
3. set of solutions already found (NIL if none is found)

The solutions are each a pair: a list of added literals, followed by a binding list.

The arguments to a CPROC are in internal form (the LISPish way to write a clause without commas). CPROC's return either:

1. NIL -- don't apply the coercion; or
2. TRUE -- use the coercion; or
3. a number -- which becomes an additional upper bound on depth of coercions used. Thus if a CPROC returns 0 then the coercion will be applied but no coercions will be used below it.

CPROC's often will use the function GROUND(X) which is true if x is a ground expression, CONSTANT(X) which is true if x is a constant, and the global variable UNIFDEPTH which gives the current recursive level inside unification. UNIFDEPTH = 0 at the top.

Some typical CPROC's are:

- * only use the coercion if no other solutions have been found
 UNIQUE = PROG(N,P,SOL), SOL EQ NIL END;
- * only use the coercion at the top level arguments
 TOP = PROC(N,P,SOL), UNIFDEPTH EQ 0 END;
- * only use the coercion if one of the two expressions is ground
 ONEG = PROC(N,P,SOL), GROUND(N) OR GROUND(P) END;

A rather odd way to define a CPROC is:

C CPROC PROC(N,P,SOL), SOL EQ NIL END;

Here we have defined the CPROC and associated the coercion in one statement.

A number of global variables affect just how coercions will be applied.

- BVARS if true don't apply any coercions whenever both terms are variables. Normally this is true.
- ONEWAY if true apply the coercions in one direction. That is, if C COER L,R is a coercion only try to use L with the item from the negative clause and R with the item from the positive clause. Normally this is false.
- MAXCOERS the search for a unifier uses this value as a depth bound. 0 would prevent any coercions from being used. The switch can be used to turn DILEMMA into a standard input resolution theorem prover.

The actual algorithm used to apply coercions is given below:

To unify N and P with the coercion C COER L,R; :

- (1) If N and P are variables, and BVARS is true,
bind N to P and return
- (2) If POSSIBLE(N,L) and POSSIBLE(P,R) then
 - a) if there is a CPROC associated with C execute it;
return false if it returns false
 - b) otherwise, if the current depth is less than MAXCOERS
compute the set of unifiers which simultaneously
satisfies UNIFY(N,L) and UNIFY(P,R).
- (3) If ONEWAY is true, then return; otherwise, apply (2)
to the coercion COER R,L; and add any new solutions
found.

In the above, POSSIBLE(X,Y) is true if
X is a variable or Y is a variable or X and Y start
with the same function symbol.

3.4 PARTIAL EVALUATION

Sometimes we know exactly how to solve a problem or some part of a problem. For instance we could have a particular interpretation which suggests values of functions. Suppose we have a predicate

$$P(x,y,z) \leftrightarrow x \cdot y = z$$

If x , y , and z are all integers, we may check this predicate in a model, the computer hardware arithmetic. The check is done by the following sequence: "multiply x times y and compare with z ". This sequence is an imperative explaining how to check items.

If, on the other hand, x were a variable we would like to execute the following: if y and z are integers and $y \neq 0$ then

$$\text{set } x = z \cdot y^{-1}$$

In addition, once a resolvent is computed, there are often simplifications to be made. For example, we could represent items in a canonical form. Or, we could evaluate functions, e.g. $S(S(0))$ goes to 2. At the same time we may add heuristic functions.

Partial evaluation is a technique of associating functions written in a standard computer language with predicates and Skolem functions. The collection of lower level functions is called the computational base. Whenever we know how to achieve some part of a goal in a simple way we may add a computational

function using all the standard tricks of the computer programmer.

We may associate with each N place function $N+1$ BALM procedures. Each time we bind a variable which occurs inside a function we will execute the associated procedures. The first N of these procedures corresponds to the arguments of the function. They are executed only if the argument is a variable. The output of each procedure will be a new binding for the variable. The last procedure is associated with the function as a whole. It will return a set of bindings and a value which is to replace the function. Each of these procedures gets the function as its argument. A typical function would be the successor function $S(n)$. It is mapped to $n+1$ if n is an integer, and otherwise it returns the symbolic quantity $S(n)$.

In a similar way, we can associate a BALM procedure with each predicate. The result of executing such a procedure should be either a replacement literal, TRUE indicating that the literal may be assumed true or FALSE indicating that the literal cannot be proved. These procedures may also bind variables.

Several previous attempts have been made to use partial evaluation. Applied only to ground clauses and then only allowed to return a truth value, ad hoc partial evaluations could be constructed easily. However, when the attempt was made to generalize to some variant of the scheme we have

suggested, a problem appeared. Suppose we have a successor function $S(N)$ where for instance $S(S(0))$ evaluated to 2. The difficulty occurs when we have to unify 2 and $S(N)$. Since no binding will make 2 and $S(N)$ identical, they could not be unified. Yet for completeness we certainly need them to be. The coercion technique provides a way of retaining these unifications. All we need do is find conditions under which 2 and $S(N)$ are equivalent. We could for example use the coercion:

SUC COER X, S(N), EQ(X, PLUS(N,1)) .

A second use of partial evaluation is to adjust the global search strategy. In the course of executing a procedure we may change a number of global variables. These delay the expansion of certain nodes or cause others to be processed immediately. The idea is that low level computations done inside of partial evaluation can be used to direct the search strategy. If, for example, we had a predicate $INT(x,y,z)$ to stand for "integrating x with respect to y gives z" we could have a clause of the form

$INT(PLUS(P,Q),Y,PLUS(R,S)) \leftarrow INT(P,Y,R) \wedge INT(Q,Y,S)$

If we wanted to use this rule only when P was simpler than $PLUS(P,Q)$ we could add an additional predicate:

$INT(PLUS(P,Q),Y,PLUS(R,S)) \leftarrow INT(P,Y,R) \wedge INT(Q,Y,S)$
 $\wedge SIMPLEFORM(P,PLUS(P,Q))$

$SIMPLEFORM$ could reject terms of a complex nature by returning FALSE if P or $PLUS(P,Q)$ were too complex.

By embedding heuristics within the partial evaluation scheme our system functions rather differently than other AI languages. In a language like PLANNER, a clause is represented by an ordered sequence of pairs. Each pair consists of a literal and a filter.

Prior to resolving on a given literal, the filter is called to produce an ordered sequence of clauses to resolve with. Often the filter acts as a generator producing the elements of this sequence one at a time. Resolution is then attempted only between the literal and the first clause in the sequence. An automatic backup system allows the program to eventually compute the other resolvents. In our system, a clause consists of a set of literals and strategies. When we have selected a literal, we resolve it with all other possible clauses. We put all strategies which were in either parent clause into the resolvent. Finally we execute all of these strategies. Besides setting global variables, the strategies may return TRUE, meaning delete this strategy from the resolvent, FALSE, meaning delete the resolvent, or another strategy to be put into the resolvent. If a strategy returned a pointer to itself, the strategy would be invoked again for descendants of the resolvent.

The PLANNER scheme has the advantage that if the filter is reasonable, it can cut down the number of attempted resolutions. On the other hand, we see a number of advantages to our system. First, since the strategy obtains the value of

variables after the resolvent has been computed it may make a more informed decision. Second, all the strategies are evaluated at each point rather than just one. Third, since strategies may remain in the descendants, a strategy may wait in making a decision until additional information is available. Finally there are many more possibilities for parallelism since we compute all the resolvents at once, and then evaluate the strategies simultaneously. Perhaps we should add one more difference: suppose a literal l could be advantageously resolved with a literal in a clause C . In PLANNER the recognition of that advantage would be computed by a filter associated with the literal l . In our language, the recognition occurs by a strategy associated with C . If we add a new clause, we need only put the strategy in that clause. In PLANNER we would have to change the filters associated with all old literals which could resolve with this clause. Thus, in Dilemma, as we add new clauses, telling of better ways to do things, less changes need to be made in the original clauses.

3.4.1 PARTEVAL

The PARTEVAL statement is used to associate BALM procedures with the partial evaluation of function and predicate symbols. These procedures which we will call PE's are executed just after the resolvent is computed.

```
<pred symbol> PARTEVAL <proc>;
```

```
<func symbol> PARTEVAL <proc>;
```

The argument of a PE is the part of the literal which starts at the predicate or function symbol. It is almost in internal form. However, variables are changed to the BALM id's

X1,X2,X3,...,X12. If we have

F PARTEVAL FPROC;

P PARTEVAL PPROC;

and the resolvent contains the literal (P (F l) (A)) then the execution operates as follows. First FPROC will be called with argument (F X1). If FPROC returns its argument, PPROC will be called with (P (F X1) (A)) as its argument. If a PE is used for a function, it must return a replacement expression (one possibility is its argument). To reject a clause the function need only set the global variable RETAIN to FALSE.

PE's used for predicates may return

1. TRUE -- delete the literal
2. FALSE -- delete the clause
3. replacement literal.

For PE's which are to be associated with predicates an alternative diction is available:

<pred> PART <i> = <proc>; $1 \leq i \leq 6$

In this form the PROC is called only if the i^{th} argument of the literal is a variable. It may return:

1. TRUE -- delete the literal
2. FALSE -- delete the clause
3. A binding for the variable which will make the literal true
4. A replacement literal.

PE procedures generally consist of a few calls to built-in functions. We will first describe several of these functions in detail and then later give a complete list.

The built-in function BIND(exp1,exp2) is used to assign values to Dilemma variables; exp1 is replaced by exp2. One of the odd properties of BIND is that anything may be bound to anything else. If we bind (F X1) to (G X1) then all (F X1) expressions in the clause are changed to (G X1). Also if we bind a variable other than X1,X2,...,X12 to a value that fact is recorded on a special list which is associated with the clause. PE's may use BIND to communicate. The function ISBOUND(ID) finds out if ID is bound or not. The result of ISBOUND is TRUE or FALSE. The global variable BNDVAL gets the bound value. CRACK is a generalized assignment, e.g. CRACK(pattern,expression). Its first argument is a pattern. This is matched with the second argument; structure and constants must match. Whenever a variable is matched with a subexpression, an assignment is made. An additional Dilemma statement allows variables to be typed:

```
TYPE <id> = <proc>;
```

Typed variables are global. Prior to CRACK assigning a value to a typed variable, the procedure is called with the value as its argument. The assignment is carried out only if the procedure returns true. CRACK itself returns a value: TRUE, if the pattern matches the expression and all assignments could be carried out; and FALSE otherwise. Suppose ARG = '(P (1) (2) (3)). Then CRACK(' (T X Y Z), ARG) would be true and would set the values of T to 'P, X to '(1), Y to '(2), and Z to '(3).

If we had said TYPE X = INTQ; TYPE Y = INTQ; TYPE Z=INTQ;
the assignments would be T = 'P, X = 1, Y = 2, and Z = 3.
If ARG2 = '(P X1 3 4) then CRACK (' (T X Y Z), ARG2)
would be false.

Suppose we want to do addition by means of a PE.
We could have a predicate ADDSUP(X,Y,SUM) which is
true if $X + Y = \text{SUM}$. If X, Y and SUM are all numbers,
say 1, 2, 5, then a PE associated with ADDSUP would be called
with (ADDSUP (1) (2) (5)) as its argument.

We could type the arguments as above. Then CRACK
would set X = 1, Y = 2, Z = 5. So the addition PE would be

```
ADDSUP = PROC(PRED),  
      IF NOT(CRACK('(T X Y Z), PRED)) THEN PRED  
      ELSE (X + Y) EQ Z END;
```

3.5 Completions

Suppose we had a negative clause $\neg(P(x) \wedge Q(x))$ and resolve away the $P(x)$ in a series of steps. If x receives the final value $f(g(a))$, we may summarize the closed subtree by the positive literal $P(f(g(a)))$.

We will call such a positive literal the completion of the associated negative literal. In a sense, the completion of a literal is a lemma of the proof. The lemma procedure of model elimination [4] finds lemmas in this way. Since each completion is a new unit clause implied by the set of clauses, we may add the completions to the original set. Furthermore, since these units have already been generated by the proof, resolutions involving these units are likely to be useful. To that end, when we produce a completion we enter a unit resolution mode attempting to produce new clauses specialized to the problem. For example, suppose we try to prove two lines equal by showing that corresponding triangles are congruent. When we remove the literal for convergence we may resolve the completion with clauses of the form

CONGRUENCE \rightarrow EQLINE

CONGRUENCE \rightarrow EQARG

to derive a set of units saying that corresponding points of congruent triangles are equal. Of course, completions are able to generate a lot of additional clauses. In the implementation a number of processes are available to cut down

completions. These processes allow us to associate BALM procedures with predicate symbols. Whenever a literal is completed the BALM procedure associated with its predicate symbol is called. This procedure may return:

- (1) TRUE -- add the completion to the input clauses
- (2) FALSE -- don't add it
- (3) A list of names of other clauses -- the system will do unit resolutions with these clauses and any of their descendants produced by this or earlier completions.

3.5.1. Completion Syntax

The statement

`<proc> COMPLETES <pred>;`

causes procedure `<proc>` to be called when a positive unit with the predicate symbol `<pred>` is produced. The procedure receives the unit as the argument. The procedure may return

- (1) TRUE -- add the unit to the positive clauses
- (2) FALSE -- don't add the unit
- (3) A list of names of theorems. This is interpreted to mean:
do one step of unit resolution with the unit
on each of these theorems and on any dependents
of the theorems produced by this or earlier
completions. (An empty list of names acts like FALSE).

Effectively completions add a unit resolution component.

When completions are provided each clause may contain literals of the form `(MARK x)`, where `x` is a literal previously resolved upon.

3.6. How to Control the Search

Clauses are selected in complexity order (lowest merit first). The procedure COMPLEX(x) computes the complexity of the list of literals x.

The function SELECT1(x) returns one of the literals in the list of literals x. This will be the next literal resolved on.

The built-in functions COMPLEXITY and SELECT1 are defined as

COMPLEXITY(x) ↔ number of symbols in x

SELECT1(x) ↔ return the hd of x.

A number of functions are provided which may be used when writing new versions of these procedures. (Many of these functions are similar to standard BALM functions but use EQUAL rather than EQ.)

DELETE(item,list)	returns list with item removed. changes list in place.
VARIABLE(item)	returns true if item is a variable e.g. an atom which is not a constant
LOOKUP(item,list)	returns the value associated with item on list. False if item is not on list.
MEMBER(item,list)	returns true if item is on list
SETPROP(id,prop-name,value)	sets the property prop-name of variable id to value.
POSLIT OF name	returns a copy of the positive literal of a clause name.

Also three global variables are of interest when writing these functions.

NEGS	all contexts not yet resolved on
DONE	all contexts already used
NOTEQLIST	ground pairs known not to be unifiable.

3.7 Built-In Equality

The clause

REFLEX ASSERT EQ1(x,x);

is built-in. The clause

THEOREM EQ1(x,y) FROM EQ2(x,z) \wedge EQ2(z,y);

is built-in. EQ1 and EQ2 are unifiable (however a coercion must be used in the unification).

3.8 System Parameters

Users may wish to change the following control variables:

CLAUSEPRINT	print each resolvent if true
NEGPRINT	print each negative clause before finding possible resolvents
MAXLITS	maximum length of a clause
MAXLEVEL	maximum depth of a clause (0 is the depth of an input clause)
MAXCOERS	maximum number of coercions used in 1 unification
MAXCPX	maximum complexity value

MAXNEST	maximum number parentheses of a clause
DEBUG	If true the system will print intermediate results; this is for internal debugging of the system rather than of DILEMMA programs. However, it often will prove useful in removing some of the trickier bugs.

3.9 A Complete Example -- Missionaries and Cannibals Using PE Functions

As an example, we will give a program for the usual missionary and cannibal problem. Suppose there are 3 missionaries and 3 cannibals on one side of a river, and a boat which holds at most 2 people. We wish to find a sequence of moves which gets everyone across the river without ever having a group of missionaries outnumbered.

We will use the following predicates.

RIGHT(c,m) \leftrightarrow the problem may be solved with C cannibals,
M missionaries and the boat on the right

INRANGE(X) \leftrightarrow X is less than or equal to the number of
people who can be on the boat

$$\text{LTE1}(x, y) \quad \leftrightarrow \quad x < y$$

OK(C,M) ↔ it is OK to have C cannibals and M missionaries together

$$\text{OKBOAT}(x) \leftrightarrow 0 < x < 2$$

LEFT(C,M) \leftrightarrow equivalent of right with C cannibals,
M missionaries, and the bota at the left.

BEST(X,Y) \leftrightarrow a heuristic predicate

EQL(x,y) $\leftrightarrow x = y$

and the following functions:

PLUS(x,y) $\leftrightarrow x + y$

MINUS(x,y) $\leftrightarrow x - y$

The DILEMMA program for solving this problem is as follows:

CONSTANT 0,1,2,3,4,5,6 END;

RT THEOREM RIGHT(CX,MY) FROM

INRANGE(X) \wedge INRANGE(Y) \wedge LTE1(X,CX) \wedge LTE1(Y,MY)

\wedge OK(PLUS(MINUS(3,CX),X), PLUS(MINUS(3,MY),Y))

\wedge OK(MINUS(CX,X), MINUS(MY,Y))

\wedge OKBOAT(PLUS(X,Y) \wedge LEFT(PLUS(MINUS(3,CX),X), PLUS(MINUS(3,MY),Y))

\wedge BEST(PLUS(MINUS(3,CX),X), PLUS(MINUS(3,MY),Y));

LT THEOREM LEFT(CX,MY) FROM

INRANGE(X) \wedge INRANGE(Y) \wedge LTE1(X,CX) \wedge LTE1(Y,MY)

\wedge OK(PLUS(MINUS(3,CX),X), PLUS(MINUS(3,MY),Y))

\wedge OK(MINUS(CX,X), MINUS(MY,Y))

\wedge OKBOAT(PLUS(X,Y) \wedge RIGHT(PLUS(MINUS(3,CX),X), PLUS(MINUS(3,MY),Y))

\wedge BEST(MINUS(3,CX), MINUS(3,MY));

INZERO THEOREM INRANGE(X) FROM EQL(X,0);

INONE THEOREM INRANGE(X) FROM EQL(X,1);

INTWO THEOREM INRANGE(X) FROM EQL(X,2);

EQL ASSERT EQL(X,X);

```

TYPE XV = INTQ; TYPE YV = INTQ;
LTEL PARTEVAL PROC(PRED);
    IF NOT(CRACK(=(T XV YV),PRED)) THEN PRED ELSE XV LE YV END;
MINUS PARTEVAL PROC(X FN),
    IF NOT(CRACK(=(T XV YV),FN)) THEN FN ELSE XV-YV END;
PLUS = PROC(FN),
    IF NOT(CRACK(=(T XV YV),FN)) THEN FN ELSE XV+YV END;
PLUS PARTEVAL PLUS;
OKA THEOREM OK(C,M) FROM LTEL(C,M);
OKB ASSERT OK(C,0);
OKBOAT THEOREM OKBOAT(X) FROM LT1(X,3)  $\wedge$  LT1(0,X);
LT1 PARTEVAL PROC(FN), IF NOT(CRACK(=(T XV YV),FN)) THEN
    FN ELSE XV LT YV END;
* BEST ADJUSTSTHE HEURISTIC VARIABLE MERIT
BEST PARTEVAL PROC(PRED), BEGIN( ),
    IF NOT(CRACK(=(T XV YV),PRED)) THEN RETURN PRED,
    MERIT = -(XV + YV), RETURN TRUE
    END END;
* THE PROVE STATEMENT
PROVE(LEFT(3,3));

```

The actual run of this example produces a 93 level proof generating some 678 resolvents.

3.10 Implementation Outline

The information in this section may be useful for writing fancy PEs and CPROCs. Internally, a clause is represented by an 8 element vector called a context, whose elements are:

- (1) id *0xx is a generated name
- (2) pointer to negative parent
- (3) pointer to literals (as a tree structre)
- (4) length in negative literals
- (5) mixed or positive parent
- (6) special bindings
- (7) depth of clause (0 is an input clause)
- (8) merit (actually complexity) of clause

The variables in a clause are represented by numbers.

Constants, function symbols, and predicates are heads of lists.

Variables are not.

3.10.1 Macros

The following set of macros is provided for accessing parts of a clause:

- (1) NAMEOF(context) - name of the clause
- (2) NEGPAR(context) - returns vector pointer to parent
- (3) LITLIST(context) - literals in internal form
- (4) CLENGTH(context) - length in terms of number of literals
- (5) MIXEDPAR(context) - returns vector pointer to parent
- (6) BINDING(context) - special bindings
- (7) DEPTHVAL(context) - depth of clause
- (8) MERITVAL(context) - merit of clause

The above set can be used as the left-hand side of an assignment statement.

3.10.2 Functions

BIND(exp1,exp2) - replace exp1 by exp2 in the clause; retain this if exp1 is a variable other than an x

ISBOUND(var) - see if this resolvent or an earlier parent had an assignment for this variable (returns true or false) (sets NBDVAL=binding value if found)

ISBOUND(var,c) - similar to ISBOUND(var) but starts search at context c

GROUND(x) - true iff x is a ground item

INPUTCL(context) - true iff context is an input clause

3.10.3 Other Details

Three global variables are often used by the PE programmer.

CURCONTEXT - current resolvent in packed vector form

DEPTH - depth of curcontext;
level 0 = input clause

MERIT - merit of curcontext;
"lowest merit first" is the selection rule.

A listing of the system, together with more detailed information on the implementation, is given in Appendix I.

3.11 Dilemma Syntax Summary

Basic Objects

names	BALM objects of type identifier
literals	
lists of literals	one or more literals separated by ^
procs	BALM objects of type code
predicate symbols	BALM objects of type identifier
function symbols	BALM objects of type identifier
constants	functions of zero arguments, or (identifier, in input form)
contexts	vector forms of clauses (used as a internal form).

Statement Types

- (1) DIFFERENT(x,y) indicates that the ground items x and y are not unifiable
- (2) CONSTANT x,y,z END; x,y,z will each be constants and are BALM identifiers
- (3) name ASSERT literal defines a positive unit clause
- (4) name THEOREM literal FROM list of literals
defines a mixed clause. The literal is positive, each of the list of literals is negative.
- (5) name COER expl,exp2; simple coercion. expl is equal (or equivalent to) exp2.

- (6) name COER expl,exp2, list of literals
compound coercion, expl equivalent
or equal to exp2 under conditions
(list of literals)
- (7) PROVE(list of literals); last statement in a program.
The list of literals is taken to be
a negative clause; may be written
as PROVE(LIST(list of lits, list
of lits, ..., list of lits));
if there is more than one initial
negative clause.
- (8) proc COMPLETES predicate; associates the procedure
with the predicate for completions
- (9) proc CPROC name; The procedure <proc> will be executed
during unification before the COERCION
<name>.
- (10) pred PART i = proc; pred will be executed after computing
the resolvent, provided the ith
argument of the predicate is still
a variable
- (11) pred PARTEVAL proc; proc will be executed just after the
resolvent has been computed.
- (12) TYPE var = proc; proc will be executed just before
an assignment is made to the var by
CRACK.

Chapter 4

The Geometry Program

We have claimed that the Dilemma programming language is extremely expressive, and that complex processes may be encoded in relatively simple ways. In order to illustrate the use of Dilemma and to give some evidence in support of this claim, we will describe a program which produces proofs for a small subset of elementary geometry theorems. The subset includes the equality of angles and of the length of lines, the parallelism of lines, and the congruence of triangles.

Gelernter [10] wrote the first program which could produce proofs for geometry. He introduced the notion of reasoning backwards. Here, one starts with the theorem, replaces it by the conditions which imply it, and then, in turn, replaces those conditions by their antecedents. This process is continued until the conditions have been replaced by axioms. Backward reasoning, which has also been called "Goal Directed Reasoning", is always used by Dilemma programs except when a completion is being applied.

Gelernter also introduced three major heuristics. Each of these involved the use of a semantic model. The program was given a "picture" in the form of coordinates of each point. In the first heuristic, goals would be rejected if they were false in this picture. Additionally, if the goal

contained a free variable, then the program would replace the goal by the set of instances using only the constants in the picture. False instances would immediately be rejected. This pruning heuristic trimmed the search tree from 1000 alternatives at each stage to five. Our program will allow free variables to remain in goals; since we don't need to pick particular instances, each clause will represent a number of different alternatives. The rejection heuristic would be successful only if every alternative could be rejected, at once. It will turn out that we do not need to use an analytic geometry model to reject goals.

The second use of a model was to evaluate the truth of "self-evident" statements. Such statements include, for example, facts about the ordering properties of points on a line, or the intersection properties of lines on a plane. If the statement was true in the model, it was assumed "proved by observation". In the Dilemma program, these "self-evident" truths are computed by partial evaluation. A third use of the model was to identify a given angle or line with all of its names, and to assume equality between any two different names of the same object. We use coercions to allow the alternative names of lines and angles to be equivalent.

The second major heuristic introduced by Gelernter was a point construction routine. When all else failed, the program would pick two points in the picture and draw the line between them. This line would then be extended to all of its inter-

sections with the other lines of the picture. Each point of intersection would become a new constant. Finally the program would start again. Our program will do constructions by unification, a way which we believe more natural.

Syntactic symmetry was the last of Gelernter's heuristics. Here, one associated a function σ with each model. Two goals would be considered variants of each other if σ applied to the first gave the second. The program would need to prove only one variant of each goal. The Dilemma notion of alphabetic variant is certainly much weaker. It is not clear just how often additional clauses could be removed if syntactic symmetry had been included.

Gelernter's program was quite long; about 20000 lines of code. The examples G1 through G5, which we will describe below were all solved by it. Several researchers attempted to clarify how these heuristics worked. It has been felt that the model heuristics were highly efficient, that they could be used in areas other than geometry, and that they were necessary to backward reasoning.

Goldstein [13] wrote a program for geometry theorem proving based on Gelernter's. Goldstein's system called BTP (Basic Theorem Prover) again was goal directed. He tried to develop a high-level, "natural" formalism to represent geometry knowledge as programs. He introduced the idea of canonical naming. Great simplifications would result if lines could have unique images. His system worked as follows:

Points could be ordered by their position in a diagram. Lines would be denoted by storing their end points in increasing order. Angle ABC would imply that A was less than C. Goldstein's program was also based on an analytic geometry model. It was written in a somewhat modified MICRO-PLANNER. Here, the simple automatic backup mechanisms proved to be an obstacle. First it was necessary to program explicit checks for duplicate goals. Second, automatic backup requires that any search be bounded. Thus his program could not create new points. Finally it was necessary to give up completely on one goal before trying an alternative. The problems G0 through G4 were solved by the BTP program.

Goldstein then proposed a second program to be called PTP (the Plausible-move-generating Theorem Prover). This program, which was not actually implemented, would do constructions in the following way: Whenever a clause involving a new point occurred, a plausible construction procedure would be used to find a new point meeting the conditions specified by the clause. Goldstein's idea was that a special procedure could be written for each type of construction and each clause that introduced a new point. Unification would select the correct clause, and thereby one or more useful move generators.

Nevins [28] tried to clarify the use of a model by writing an additional geometry program. He argued that the essential information held in a model was not numeric; in fact,

that it was the canonical ordering, the equivalence of names, and information on what looked parallel. His program ordered points lexicographically. No diagram was required. Nevins felt that the use of a model had forced previous researchers to write goal-directed programs. In his system, he included both forward and backward reasoning. The heart of his technique was a separately written LISP procedure for each paradigm of geometry. This program, which is perhaps the most efficient geometry theorem prover, has no provision for generating new points. The only aspects of a model which are used are nonnumeric. There is a list of lines which look parallel. Our own system is highly goal-directed, and is designed to construct new points through unification.

4.1 Basic Objects and Their Intended Meanings

We have 11 constants. RTANG and STANG represent constant angles. RTANG is 90 degrees, STANG is 180°. We have nine constants intended to represent points: E, P, K, N, M, C, D, Q, and B. We will introduce a total ordering on these constants by $E < P < K < N < M < C < D < Q < B$. This particular arrangement is produced by the BALM IFROMID primitive.

Only three logical functions are required. The first, $L(X,Y)$ is intended to map pairs of points to lines. The second $A(X,Y,Z)$ is intended to map triples of points to angles. Using partial evaluation we will associate canonical ordering procedures with these functions. The third Skolem function, $MIDPT(X,Y)$ is intended to mean the point midway between X and Y.

The predicates allowed in this system are:

- (1) $CONG(X,Y,Z,U,V,W)$ \leftrightarrow triangle XYZ is congruent to triangle UVW
- (2) $EQLINE(S,T)$ \leftrightarrow the length of line S is equal to that of line T
- (3) $EQANG(X,Y)$ \leftrightarrow the size of angle X equals that of angle Y
- (4) $PARALLGM(X,Y,Y,Z)$ \leftrightarrow X,Y,U,V are the corners of a parallelogram

- (5) COLIN(X,Y,Z) ↔ points X,Y,Z are collinear in that order.
- (6) PARALLEL(S,T) ↔ line S is parallel to line T
- (7) DOWN(X,Y,Z,U,V,W) ↔ a heuristic predicate defined below
- (8) DOWN30(X) ↔ a heuristic predicate defined below
- (9) DIFF3(X,Y,Z) ↔ X, Y, and Z are distinct points.

4.2 The Logical Axioms

We will now describe the clauses containing geometric information. This is the information given to the outer theorem prover.

Segment Equality

- (1) a line is always equal in length to itself.
LREF ASSERT EQLINE(X,X);
- (2) Two segments are equal if they both equal a third.
LTRANS THEOREM EQLINE(X,Y) FROM EQLINE(X,Z) ^ EQLINE(Z,Y)
^ DOWN30(X);

The introduction of a new line through transitivity is generally a poor way to do a proof. The predicate DOWN30 will make a clause 30 points more complex. This is one of the ways in which a predicate may be used to effect the search strategy.

- (3) Sums of equal segments are equal.
LPART THEOREM EQLINE(L(X,Y), L(U,V))
FROM EQLINE(L(X,R), L(U,S)) ^ EQLINE(L(R,Y), L(S,R))
^ COLIN(X,R,Y) ^ COLLN(U,S,R);

This axioms corresponds to the following picture:



$XY = UV$ if $XR = US$ and $RY = SV$



- (4) Corresponding parts of congruent triangles are equal.

LONG THEOREM $EQLINE(L(X,Y), L(U,V))$ FROM

$CONG(Y,Z,X,R,W,U) \wedge DIFF3(X,Y,Z) \wedge DIFF3(U,V,W)$

This theorem may construct new points Z and W. The DIFF3 predicates prove that these added points really form triangles.

- (5) The sides of an isosceles triangle are equal.

LISOC THEOREM $EQLINE(L(X,Z), L(Y,Z))$ FROM

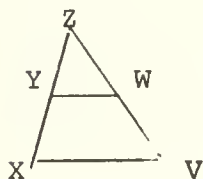
$EQANG(A(Z,X,Y), A(X,Y,Z));$

- (6) A line parallel to the base of a triangle, which bisects one side, bisects the other.

LPARBASE THEOREM $EQLINE(L(X,Y), L(Y,Z))$ FROM

$EQLINE(L(Z,W), L(W,V)) \wedge PARALLEL(L(Y,W), L(X,V))$

$\wedge COLIN(X,Y,Z) \wedge COLIN(Z,W,V);$



if $YW \parallel XV$ and $ZW \parallel WV$,
then $XY = YZ$

- (7) Midpoints bisect lines.

LMID ASSERT $EQLINE(L(X,MIDPT(X,Y)), L(Y,MIDPT(X,Y)));$

Additional ways in which lines may be proved equal are found in the section on coercions.

Angle Equality

(8) An angle is always equal to itself.

AREFLX ASSERT EQANG(X,X);

(9) Two angles are equal if they both equal a third.

ATRANS THEOREM EQANG(X,Y) FROM

EQANG(X,Z) \wedge EQANG(Z,Y) \wedge DOWN30(X);

Again we will use DOWN30 to lower the use of transitivity.

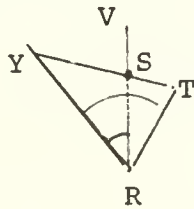
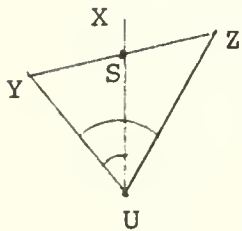
(10) Differences of equal angles are equal.

APART THEOREM EQANG(A(X,Y,Z), A(V,R,I) FROM

EQANG(A(X,U,Y), A(V,R,W)) \wedge EQANG(A(Y,U,Z), A(W,R,T))

\wedge DOWN30(X) \wedge COLIN(U,J,X) \wedge COLIN(R,S,V)

\wedge COLIN(Y,J,Z) \wedge COLIN(W,S,T)



if $\angle YUZ = \angle WRT$ and $\angle YUX = \angle VRW$
then $\angle XUZ = \angle VRT$

(11) Corresponding points of congruent triangles are equal.

ACONG THEOREM EQANG(A(X,Y,Z), A(U,R,W)) FROM

CONG(X,Y,Z,U,V,W) \wedge DIFF3(X,Y,Z) \wedge DIFF3(U,V,W);

Once again DIFF3 is used to make sure that we have two triangles.

(12) The base angles of an isosceles triangle are equal.

AISOSC THEOREM EQANG(A(X,Y,Z), A(Y,X,Z))

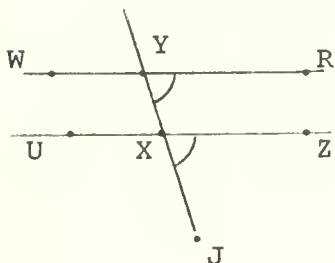
FROM EQLINE(L(X,Z), L(Y,Z));

(13) Corresponding angles of parallel lines are equal.

ACOR THEOREM EQANG(A(R,Y,X), A(Z,X,J))

FROM PARALLEL(L(W,R), L(U,Z))

A COLIN(W,Y,R) \wedge COLIN(U,X,Z) \wedge COLIN(Y,X,J);



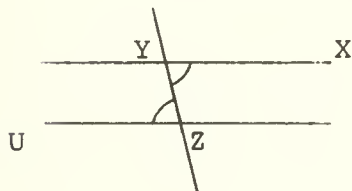
if $WR \parallel UZ$.

then $\angle RYX = \angle ZXJ$

(14) Alternate interior angles of parallel lines are equal.

ALTER1 THEOREM EQANG(A(X,Y,Z), A(U,Z,Y)) FROM

PARALLEL(L(Y,X), L(U,Z))



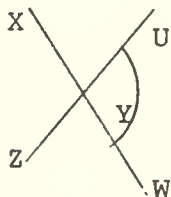
IF $XY \parallel UZ$ then

$\angle XYZ = \angle UZY$

(15) Vertical angles are equal.

VERTICAL THEOREM EQANG(A(X,Y,Z), A(U,Y,W))

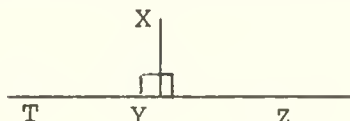
FROM COLIN(X,Y,W) \wedge COLIN(U,Y,Z)



(16) Two right angles add up to a straight line.

RTEQ THEOREM EQANG(A(X,Y,Z), RTANG) FROM EQANG(A(T,Y,X), RTANG)

\wedge COLIN(T,Y,Z)



Congruence of Triangles

(17) A triangle is always congruent to itself.

CREF ASSERT CONG(X,Y,Z, X,Y,Z);

(18) Congruence by transitivity.

CRANS THEOREM CONG(X,Y,Z,U,V,W) FROM
CONG(X,Y,Z,P,Q,R) \wedge CONG(P,Q,R,U,V,W)
 \wedge DOWN(X,Y,Z,U,V,W);

(19) The common ways to prove congruence, side-angle-angle:

CSAA THEOREM CONG(X,Y,Z,U,V,W) FROM
EQLINE(L(X,Y), L(U,V)) \wedge EQANG(A(X,Y,Z),A(U,V,W))
 \wedge EQANG(A(Y,Z,X), A(V,W,U));

(20) Side-side-side

CSSS THEOREM CONG(X,Y,Z,U,V,W) FROM
EQLINE(L(X,Y), L(U,V)) \wedge EQLINE(L(Y,Z), L(V,W))
 \wedge EQLINE(L(X,Z), L(U,W));

(21) Side-angle-side

CSAS THEOREM CONG(X,Y,Z,U,V,W) FROM
EQLINE(L(X,Y), L(U,V)) \wedge EQLINE(A(X,Y,Z),A(U,V,W))
 \wedge EQLINE(L(Y,Z),L(V,W));

Congruence is independent of the position of the viewer.
Thus we may turn the page over or around. We need to
include generators of the dihedral group to allow all
possible ways of proving congruence.

(22) CPERM THEOREM CONG(X,Y,Z,U,V,W)
FROM CONG(Y,Z,X,V,W,U);

(23) CFLIP THEOREM CONG(X,Y,Z,U,R,W)
FROM CONG(X,Z,Y,U,W,V);

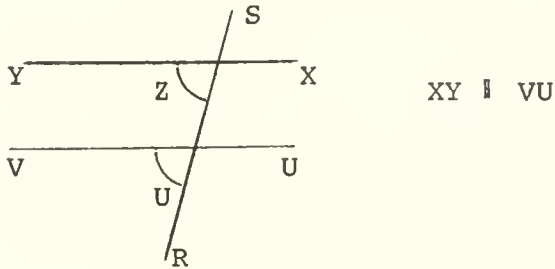
(24) The diagonal of a parallelogram, divides the
parallelogram into two congruent triangles.

CPARAL THEOREM CONG(X,Y,Z,X,U,Z) FROM
PARALLGM(X,U,Z,Y);

Parallel Line Theorems

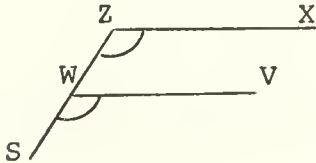
(25) Two lines are parallel if corresponding angles are equal.

P1 THEOREM PARALLEL(L(X,Y), L(U,V)) FROM
 EQANG(A(Y,Z,W), A(V,W,R)) \wedge COLIN(X,Z,Y)
 \wedge COLIN(U,W,V) \wedge COLIN(S,Z,W) \wedge COLIN(Z,W,R);



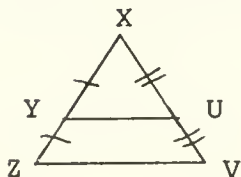
(26) We must also include the case in which the cross line doesn't pass the parallel lines.

P2 THEOREM PARALLEL(L(X,Z), L(V,W))
 FROM EQANG(A(Y,Z,W), A(V,W,S)) \wedge COLIN(Z W S);



(28) A line which bisects both sides of a triangle is parallel to the base.

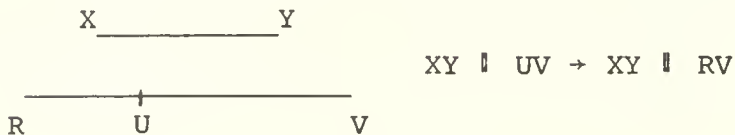
MIDPAR THEOREM PARALLEL(L(Y,U), L(Z,V)) FROM
 EQLINE(L(Z,Y), L(Y,X)) \wedge COLIN(Z,Y,X)
 \wedge EQLINE(L(V,U), L(U,X)) \wedge COLIN(V,U,X);



(28) Two lines are parallel if an extension of one is parallel to the second.

PEXT THEOREM PARALLEL(L(X,Y , L(U,V))

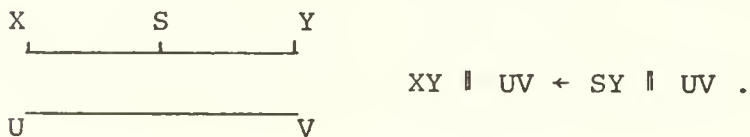
FROM PARALLEL(L(X,Y), L(R,V)) \wedge COLIN(R,U,V);



(29) Two line segments are parallel if a contradiction of one is parallel to the second. This clause is required since collinearity implies an order.

PBASE THEOREM PARALLEL(L(X,Y), L(U,V))

FROM PARALLEL(L(S,Y), L(U,V)) \wedge COLIN(X,S,Y);



(30) Two lines are parallel if they are both parallel to a third.

PTRANS THEOREM

PARALLEL(X,Y) FROM PARALLEL(X,Z) \wedge PARALLEL(Z,Y);

Since lines are often proved parallel using transitivity, we don't need to include DOWN30.

These 30 clauses represent the geometric knowledge given to the outer theorem prover. In any particular run, only a subset of these clauses is used.

4.3 Coercions

The inner theorem prover is given the following coercions:

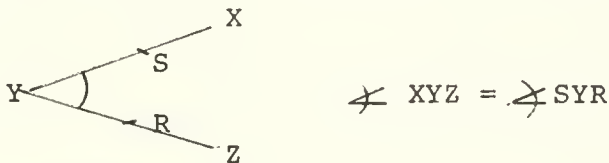
Segment equality:

- (1) LFLIP COER L(X,Y), L(Y,X);
- (2) LSYM COER EQLINE(X,Y), EQLINE(Y,X);

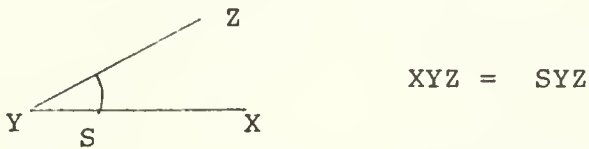
Angle equality:

- (3) ASYM COER EQANG(X,Y), EQANG(Y,X);
- (4) Alternative names correspond to the same angle.

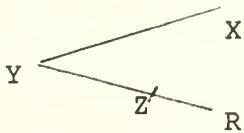
ANAME COER A(X,Y,Z), A(S,Y,R), COLIN(X,S,Y) \wedge COLIN(Z,R,Y);



- (5) ANAME1 COER A(X,Y,Z), A(S,Y,Z), COLIN(X,S,Y);



- (6) ANAME2 COER A(X,Y,Z), A(X,Y,R), COLIN(Y,Z,R);



- (7) ANGR COER A(X,Y,Z), A(Z,Y,X);

Triangle congruence:

- (8) CSYM COER CONG(X,Y,Z,U,V,W), CONG(U,V,W,X,Y,Z);

Each coercion was used in every run of the theorem prover. It is clear that the separation of clauses between inner and outer theorem prover was done in a somewhat arbitrary manner. The idea was that facts considered to be more "obvious" were given to the inner theorem prover.

4.4 Coercion Strategies

We use CPROCS to associate strategies with the coercions. These were chosen before any attempts were made to prove any theorems. The strategies were:

LFLIP: one of the two lines had to be ground (i.e. in the line $L(x,y)$ neither x nor y could be variables
LSYM: one of the two lines had to be ground
ASYM: one of the two angles had to be ground
ANAME: no other solutions were found (i.e. this coercion could only be used if no unifiers had been found either using previous coercions or in the standard way)
ANAME1: no other solutions were found
ANAME2: no other solutions were found
ANGR: no other solutions were found
CSYM: no other solutions were found

Only one level of coercion could be used inside unification ($\text{MAXCOERS} = 1$).

These coercions are intended to "undo" the effects of canonical ordering. The ordering is itself done by partial evaluation. The ordering scheme is:

variables < points < logical functions

Variables are represented by numbers. Thus any two variables have an induced ordering. The BALM IFROMID function is used to order points.

$L(X,Y)$ is ordered so that $X < Y$

$A(X,Y,Z)$ is ordered so that $X < Z$

The use of canonical ordering is the major factor in the effectiveness of this program. A clause represents the entire set of preimages of the canonical form. Often this could be as many as 64 different alternatives (a clause mentioning 6 lines). Almost all resolvents thus become duplicates. The combination of canonical ordering and coercion does a huge amount of tree pruning without removing completeness.

It is interesting to note that we are using lexical properties rather than position in a model to order items. One of the uses of a model in Gelernter's scheme is thus unnecessary here.

Another "model" property, the evolution of "self-evident" truths is done by partial evaluation. We may reject any clause containing an instance of $L(X,X)$ or $A(X,Y,X)$, since two distinct points are required for a line and three for an angle. The predicate COLIN is partially evaluated. We have a list of all points which are collinear in the original problem. COLIN applied to three points simply checks if the points are on this list. Skolem functions such as MIDPT add new elements to this list. The DIFF3 predicate is also evaluated.

4.4.1 The Major Heuristics

Several predicates and functions are used to compute heuristic information.

- L(X,Y) ⇔ if X and Y are variables increase the complexity of a clause by 10
- DOWN30 ⇔ increase the complexity of a clause by 30
- DOWN ⇔ if any of the arguments of the predicate are variables increase the complexity of a clause by 30.

The complexity of a clause is computed in the following way.

4.4.2 Complexity Function

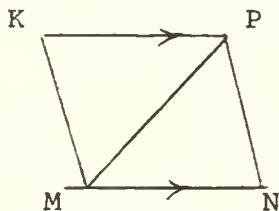
- (1) complexity starts at zero
- (2) If the clause was produced by resolving with a special hypothesis of the problem, lower the complexity by 50
- (3) For each literal which is unifiable with a special hypothesis, lower the complexity by 20;
for each other literal, increase the complexity by 10
- (4) Add twice the number of variables in each literal
- (5) Add 20 for each CONG literal
- (6) Add 5 for each nonground literal
- (7) For literals which have PE's associated to their predicate don't use 3-6; instead add $5 + 4 * \text{number of variables}$
- (8) Reject clauses with more than 3 PARALLEL predicates
- (9) Reject clauses longer than 10 literals.

4.4.3 Actual Input

Actual input to the program consists of

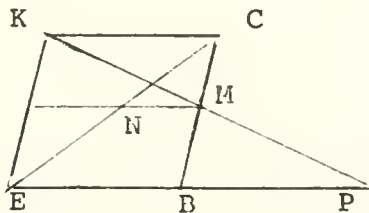
- (1) the special hypotheses of the problem
- (2) a list of collinear points as the value of the variable COLINER
- (3) a list of the names of the special hypotheses
- (4) the initial goal to be proved.

For example the G2 problem input is:



```
H1: ASSERT EQLINE(L(P,K),L(N,M));
H2: ASSERT PARALLEL(L(P,K),L(M,N));
COLINER = NIL;
HYP = (H1,H2);
PROVE(EQLINE(L(P,M),L(K,N)));
```

A somewhat more elaborate problem is G5:



```
H1: ASSERT PARALLEL(L(K,C),L(E,D));
H2: ASSERT EQLINE(L(E,N),L(N,C));
H3: EQLINE(L(K,M),L(M,D));
HYP = ' (H1 H2 H3);
```

```
COLINER = ' ( (E) (P) (K) ) ( (K) (P) (E) ) ( (P) (N) (M) ) ( (M) (N) (P) )
           ( (K) (M) (D) ) ( (D) (M) (K) ) ( (E) (B) (D) ) ( (D) (B) (E) )
           ( (C) (M) (B) ) ( (B) (M) (C) ) ( (E) (N) (C) ) ( (C) (N) (E) ) );
PROVE (EQLINE (L (E, P) , L (P, K) ) ;
```

4.5 Point Constructions

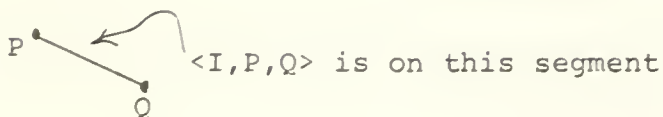
The geometry program which we have described in the previous section, lacks a very important geometric fact; that any pair of distinct nonparallel lines intersect in exactly one point. Given this fact, a program could introduce a new point defined as the intersection of a pair of lines. In this section, we will show how this fact may be added to the original system.

One possible way to incorporate this information would be to use an axiomatic approach. Here, we would define a point predicate and then add a number of clauses specifying how points and lines are related. Such an approach leads to very many new clauses. These represent information of a very different nature from that in the Sections 4.1-4.3. For example to say that a point is either to the left or right of a line is far more basic than to say that two triangles are congruent whenever corresponding sides are equal. Until now we have represented simple point ideas such as ordering or collinearity entirely by partial evaluations; we chose to encode this new fact in the same way.

We first introduce two new constants, R and I . We will write $\langle R, P, Q \rangle$ when we wish to represent an undetermined point on the line joining P and Q , with the order of the points being P, Q, X .



Similarly $\langle I, P, Q \rangle$ will represent a point inside the segment PQ:



Next we introduce a six place logical function

$\text{INTER}(\text{TYPE1}, \text{TYPE2}, P11, P12, P21, P22)$

This will represent the point which is both of the triples

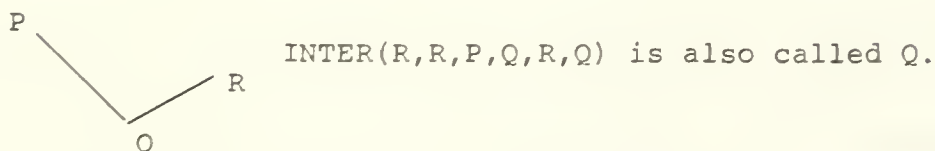
$\langle \text{TYPE1}, P11, P12 \rangle$

and

$\langle \text{TYPE2}, P21, P22 \rangle$.

If the lines $P11 P12$ and $P21 P22$ are parallel then INTER will be undefined.

The use of INTER raises a new issue: that of multiple names for the same point, e.g. in the picture:



We will therefore associate a partial evaluator with INTER, which will reduce INTER of 6 constants to a point name when this is possible. Of course we will need a coercion to unevaluate INTER. This coercion will allow

$\text{INTER}(R, X, P, Q, Y, Z)$ to unify with a point B

with the added conditions:

$X = R \wedge \text{COLIN}(Y, Z, B)$

or

$X = I \wedge \text{COLIN}(Y, B, Z)$

The COLIN predicate needs to be changed as well.

COLIN(P,Q,X) evaluates to TRUE with the condition

$$X = \text{INTER}(R,U,P,Q,R,W)$$

COLIN(P,Q,INTER(R,U,B,E,V,W)) evaluates to true with

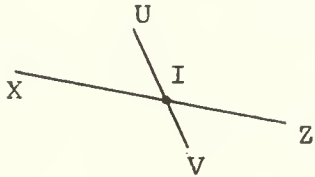
the conditions $U = R$, $V = P$, $W = Q$, PQ not parallel to BE.

The most pronounced change added to our system is the need to detect when lines are not parallel. To resolve this we will introduce additional information from the diagram. We wish to point out that this is still nonnumeric information. We will give the program a list, each of whose elements will be a list of line segments. The interpretation will be that any two segments which are on the same list "look parallel". The program will not attempt to construct a point as the intersection of two lines both of which are on the same list. Being on the same list will not be sufficient to prove parallelism. It will however be sufficient to prevent constructions. We introduce a new PE predicate $\text{CROSSES}(X,Y,U,V) \leftrightarrow XY$ does not look parallel to UV.

Finally we want to add two new clauses. Since the coercions are present, these clauses are not necessary for completeness, yet they increase the efficiency. Both clauses will state that a line is parallel to a subsegment bounded by a second line.

PCR1 THEOREM PARALLEL(L(X, INTER(TP1, TP2, U, V, X, Z)),
 L(X, Z)) FROM CROSSES(X, Z, U, V);

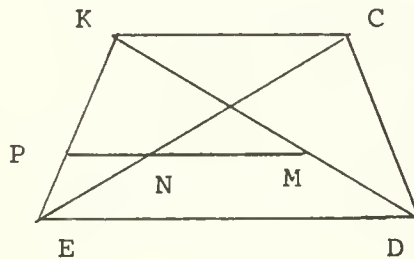
PCR2 THEOREM PARALLEL(L(X, INTER(TP1, TP2, X, Z, Y, V)),
 L(X, Z)) FROM CROSSES(X, Z, U, V);



These clauses state that
 XI is parallel to XZ,
 where I is the intersection
 of UV and XZ.

A sample input to the extended geometry program is the
 following:

Given the problem



Special hypotheses $KC \parallel ED$, $EN = NC$, $KM = MD$

Prove $EP = PK$

Computer Input:

H1 ASSERT PARALLEL(L(K, C), L(E, D));

H2 ASSERT EQLINE(L(E, N), L(N, C));

H3 ASSERT EQLINE(L(K, M), L(M, D));

[continued]

* list of names of special hypotheses

HYP = = (H1 H2 H3);

* list of collinear points

COLINER = ((E) (P) (K)) ((K) (P) (E))
 ((P) (N) (M)) ((M) (N) (P))
 ((K) (M) (D)) ((D) (M) (K))
 ((E) (N) (C)) ((C) (N) (E)));

* list of lines which look parallel

LOOKSP = ' ((K C) (N M) (P M) (E D));

* list of alternative names

PNames = ' ((R R K P D E) E)
 ((R R C N D E) E)
 ((I R E K M N) P)
 ((I I E C P M) N)
 ((R R E D K M) D));

* finally the prove statement

PROVE(EQLINE(L(E, P), L(P,K)));

The next chapter contains the results of running both the original and the extended geometry program on a number of simple problems.

CHAPTER 5

EXPERIMENTAL RESULTS

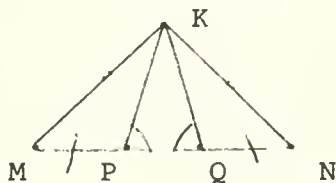
In order to test the effectiveness of the programs described in Chapter 4, we performed a number of experiments. We chose six different geometric problems, and did a number of variations of each. In this chapter, each of these experiments is analyzed in detail. For each problem we will give a picture, and a list of axioms. We will show the refutation tree and indicate where the coercion scheme eliminated resolutions.

The six geometric problems which we named $G_0, G_1, G_2, G_3, G_4, G_5$ have all been proved previously by machine.

G_0 , the simplest of the problems, was introduced by Ira Goldstein [13]. Problems G_1 through G_5 are due to Gelernter [10,11]. Examples G_0 through G_4 were run with the basic system. Variations of G_5 were run with both the basic and extended system.

5.1 The G_0 Problem (Problem 32, June 1969, N.Y. State Regents)

Given segment $MPQN$ with $MP = QN$ and angle $KPQ = \text{angle } KQP$ prove that segment $MK = \text{segment } KN$.



To specify the input to this problem, we must first indicate which clauses and coercions were used.

Coercions:

<u>ANGR</u>	}	alternative names of angles are equivalent
<u>ANAME1</u>		
<u>ANAM2</u>		
<u>CSYM</u>	}	CONG, EQLINE, EQANG are symmetric relations
<u>LSYM</u>		
<u>ASYM</u>	}	the line XY is equivalent to the line YX
<u>FLIP</u>		

Clauses:

<u>AST</u>	two angles are equal if both are straight
<u>APART</u>	parts of equal angles are equal
<u>ATPANS</u>	} angle equality is transitive and reflexive
<u>AREFLX</u>	
<u>LISOC</u>	sides of isosceles triangles are equal
<u>LCONG</u>	corresponding sides of congruent triangles are equal
<u>CSAS</u>	congruence by side, angle, side.

Input:

```

H1  ASSERT EQANG(A(K,P,Q), A(P,Q,K));
H2  ASSERT EQLINE(L(P,M), L(N,Q));
HYP = '(H1 H2);

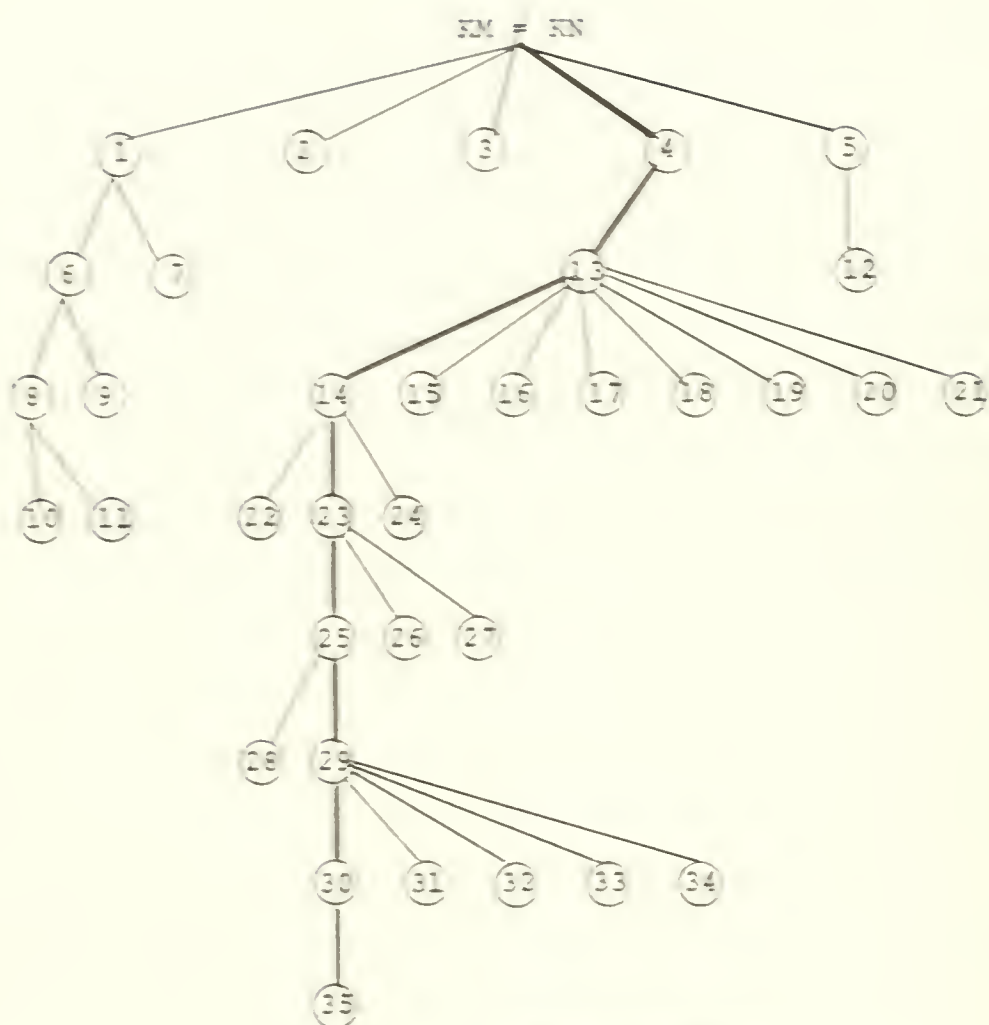
COLINER = '(((M)(P)(Q)) ((P)(Q)(N)) ((M)(P)(Q)(N))
           ((Q)(P)(M)) ((N)(Q)(P)) ((N)(Q)(P)(M)) );

PROVE (EQLINE(L(K,M), L(K,N)));

```

Total input: 8 coercions, 9 clauses.

The proof required a total of 44.7 seconds, during which time 35 resolvents were generated. Figure 4 shows the search tree. 12 clauses were selected for resolution. Each node is a clause number. The dark line indicates the proof path. The proof requires 8 out of the 12 selected clauses. While a great deal of the selectivity is due to the limited number of clauses, the accuracy is quite surprising. Figure 5 shows the refutation tree. The proof corresponds to some 23 resolutions in a standard resolution prover.



44.7 seconds

12 clauses selected

35 resolvents

Figure 4. G0 [Variation 1]. Search Tree.

The critical factor in this run of G0 is that there are few unnecessary input clauses. To test the robustness of the system we ran the following variations.

G0.1

All the 8 coercions and 9 clauses of G0

plus the clauses

AVT ↔ vertical angles are equal

LREF ↔ line reflexivity

G0.2

All 8 coercions and 11 clauses of G0.1

plus the clauses

ALTER ↔ alternate interior angles of parallel lines are equal

CSAA ↔ congruence via side, angle, angle

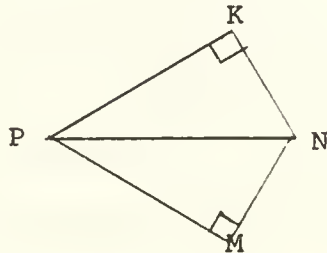
The resultant timings to find the proof were:

G0	44.7 seconds	35 clauses generated
G0.1	45.6 seconds	35 clauses generated
G0.2	52.3 seconds	40 clauses generated

Each of the three runs produced the same proof.

5.2 The G1 Problem (Gelernter, Problem 1)

Given that angle KPN = angle NPM, and that angles PKN and PMN are both right angles, prove that segment KN = segment NM.



Input to the theorem prover consisted of the 8 coercions used for G0 plus the following clauses:

AST	straight angles are equal
APART	parts of equal angles are equal
ATRANS	} angle equality is transitive and reflexive
AREFLEX	
LREF	line equality is reflexive
LISOC	sides of an isosceles triangle are equal
LCONG	corresponding sides of congruent triangles are equal
CSAA	congruence by side, angle, angle
CSAS	congruence by side, angle, side.

Only LCONG, CSAA, LREF are used in the proof.

The special input for G1 was:

```
H1  ASSERT  EQANG(A(K,P,N), A(N,P,M));
H2  ASSERT  EQANG(A(P,K,N), RTANG);
H3  ASSERT  EQANG(A(P,M,N), RTANG);
```

[continued]

HYP = ' (H1 H2 H3);

COLINER = NIL;

PROVE(EQLINE(L(K,N), L(N,M)));

Gelernter's system took approximately 25 seconds to produce a 10 step proof which is equivalent to the resolution proof. The seven step coercion proof required some 86.5 seconds selecting 12 clauses and producing 58 resolvents. Gelernter's program was run on a 7090 while the Dilemma system runs on the considerably faster CDC 6600. At the top level the proof consisted of:

EQLINE(L(K,N), L(N,M))

└─ LCONG

(5) $\neg \text{CONG}(\text{NXK NYM}) \vee \neg \text{DIFF}(\text{K N X}) \vee \neg \text{DIFF}(\text{M N Y})$

└─ CSAA

(8) $\neg \text{EQLINE}(\text{L(NX), L(NY)}) \vee \neg \text{EQANG}(\text{A(KXN), A(NYM)})$

$\vee \neg \text{EQANG}(\text{A(XKN), A(YMN)}) \vee \neg \text{DIFF}(\text{KNX})$

$\vee \neg \text{DIFF}(\text{MNY})$

└─ LREF

(35) $\neg \text{EQANG}(\text{A(KXN), A(NXM)}) \vee \neg \text{EQANG}(\text{A(XKN), A(XMN)})$

$\vee \neg \text{DIFF}(\text{KNX}) \vee \neg \text{DIFF}(\text{MNX})$

└─ H1

(46) $\neg \text{EQANG}(\text{A(PKN), A(PMN)})$

└─ ATRANS

(52) $\neg \text{EQANG}(\text{A(PKN), X}) \vee \neg \text{EQANG}(\text{X, A(PMN)})$

└─ H2

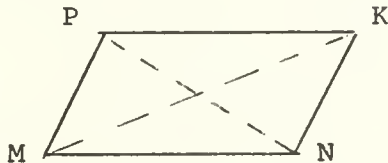
(53) $\neg \text{EQANG}(\text{RTANG, A(PMN)})$

└─ H3

(58) \square

5.3 The G2 Problem (Gelernter, Problem 2)

Given the quadrilateral PKNM with segment PK equal to segment NM and PK parallel to NM, prove segment PM = segment KN.



Inputs: the 8 coercions used in G0 and G1 plus

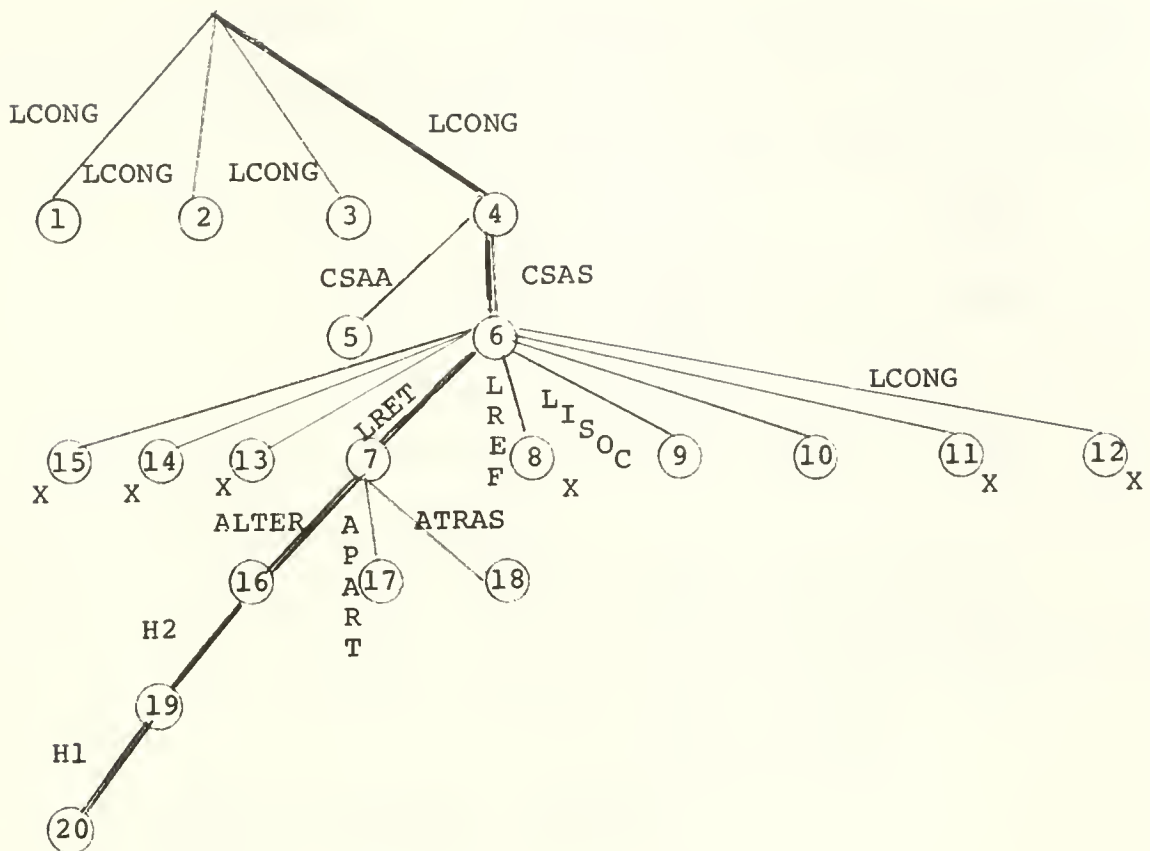
ALTER1	alternate interior angles of parallel lines are equal
AST	all straight angles are angle
APART	parts of equal angles are equal
ATRANS	angle equality is transitive and reflexive
AREF	
LREF	line equality is reflexive
LISOC	sides of an isosceles triangle are equal
LCONG	corresponding sides of congruent triangles are equal
CSAA	congruence by side, angle, angle
CSAS	congruence by side, angle, side

```
H1  ASSERT EQLINE(L(P,K), L(M,N));
H2  ASSERT PARALL (L(P,K), L(M,N));
PROVE(EQLINE(L(P,M), L(K,N));
```

The search for a proof selected 6 clauses, which produced 20 resolvents. The proof is 19 levels deep for resolution and 6 levels for coercion. The

coercion system found the proof in 10.2 seconds. This problem is by far the easiest of all the geometry examples. It is interesting to see the difference in performance between our system and Gelernter's on this problem. Since Gelernter's system did not have Skolem functions, each line segment had to be a named constant. The segment PN (the diagonal line) was not given to either program. Gelernter's program after about a minute gave up on this problem, went into its construction phase, added PN, and then after another 4 minutes produced the proof. Our program used the MK diagonal. Constructions such as PN are done automatically in our system, as the result of Skolem functions.

Another observation is that at one point our problem resolves $MK \neq KM$ with $X = X$ via the coercion $MK = KM$. This is redundant, the subtree shows that we could just as well resolve the literal away with a unary resolvent. Figure 6 gives the search tree; Figure 7 the refutation tree for G2.

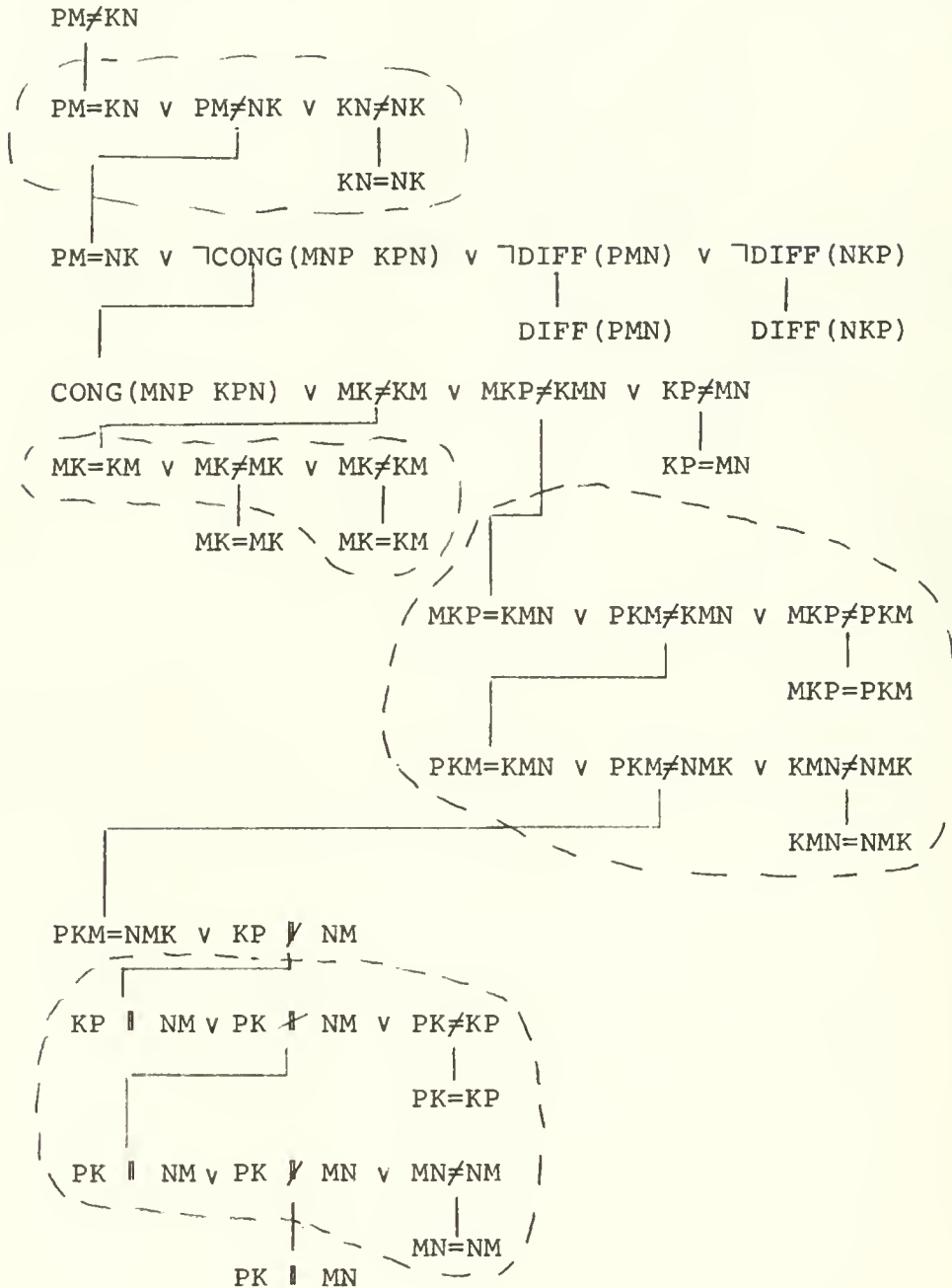


Nodes with an X below were deleted
as duplicated, too complex, or
by PES.

Figure 6. Search Tree for G2 Problem.

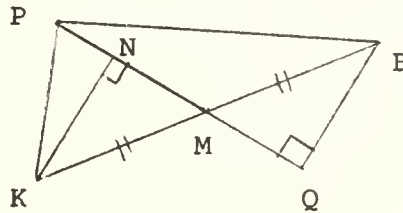
Figure 7. Refutation Graph of G2

19 resolutions vs. δ coercions. Sections in broken lines are done by the inner theorem prover.



5.4 The G3 Problem

Given the following figure, with $KM = MB$, and angles PNK and MQB both equal to 90 degrees, prove that $KN = BQ$.



Inputs: the 8 basic coercions and 10 clauses of G2 as well as:

VERTICAL	vertical angles are equal
RTEQ	two right angles form a straight angle

```
H1 ASSERT EQLINE(L(K,M), L(M,B));
H2 ASSERT EQANG(A(P,N,K), RTANG);
H3 ASSERT EQANG(A(M,Q,B), RTANG);
COLINER = '((P)(N)(M)) ((N)(M)(Q)) ((K)(M)(B))
           ((M)(N)(P)) ((Q)(M)(N)) ((B)(M)(K))';
PROVE(L(K,N), L(B,Q))
```

Gelernter ran this problem twice; first on his original system which produced a somewhat redundant proof after 8 minutes and then on an extended system which produced a proof after 1 minute. The extensions consisted of two heuristics. The first allowed goals to be expanded out of order. Identities, equalities between angles, and various other goals

could usually be satisfied in one step. Gelernter's heuristic gave preference to such goals. The second heuristic was a complexity measure.

Our system solved this problem in 4.5 minutes. It found an 8 level coercion which corresponds to a 20 level resolution proof. A total of 144 resolvents were generated from 34 selected clauses. The complexity function did rather poorly here. For comparison we give the complexity value for each resolvent in the proofs for G1, G2, and G3.

G1 Clause Value	G2 Clause Value	G3 Clause Value
5 +75	4 +75	4 +75
8 +66	6 +66	24 +66
35 +46	7 +14	51 6
46 -17	16 -38	65 +33
52 +47	19 -69	67 +77
53 -69	20 - ∞	140 -13
58 - ∞		141 -58
		144 - ∞

This table shows that the complexity measure did badly in G3 on clauses 65 and 67 which accounts for the large amount of time necessary to find the proof. Clause 67 consists of:

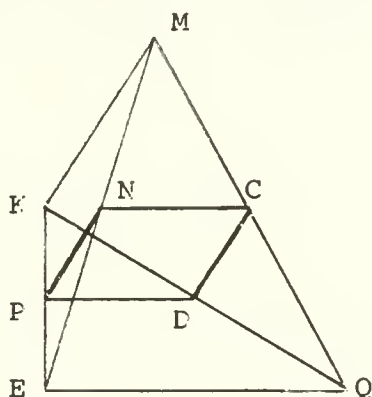
$$\neg(\text{angle}(K N M) = X \vee X = \text{angle}(M Q B))$$

This clause was produced by an application of angle transitivity which is given a penalty each time it is used.

5.5 The G4 Problem

Gelernter introduced additional extensions to his program in 1960. The first of these extensions allowed that any two names for an angle could be recognized as equal solely by using the diagram. The second extension was the point generator which has been described above. Using these extensions, Gelernter's system solved the G4 and G5 problems, requiring 1 minute on G4 and 31 minutes on G5.

G4



Given: $EP = PK$

$EN = NM$

$MC = CQ$

$KD = DQ$

Prove: $NC \parallel PD$

Inputs: the 8 coercions and 12 clauses of G3 plus

PSYM COER PARALLEL(X,Y), PARALLEL(Y,X);

PARTRANS parallelism is transitive

MIDPAR the line which bisects the sides of
a triangle is parallel to the base.

H1 ASSERT EQLINE(L(E,P), L(P,K));

H2 ASSERT EQLINE(L(E,N), L(N,M));

H3 ASSERT EQLINE(L(C,Q), L(M,C));

H4 ASSERT EQLINE(L(K,D), L(D,Q));

HYP = '(H1 H2 H3 H4);

```

COLINER = ' ( (E) (P) (K) ) ( (K) (P) (E) )
           ( (E) (N) (M) ) ( (M) (N) (E) )
           ( (M) (C) (Q) ) ( (Q) (C) (M) )
           ( (K) (D) (Q) ) ( (Q) (D) (K) ) );

PROVE (PARALLEL (L (N,C) , L (P,D) ) );

```

The proof found was 7 coercions long, and corresponded to a 20 level resolution proof. A total of 331 resolvents were produced from 32 selected clauses. Total time was 25.7 seconds.

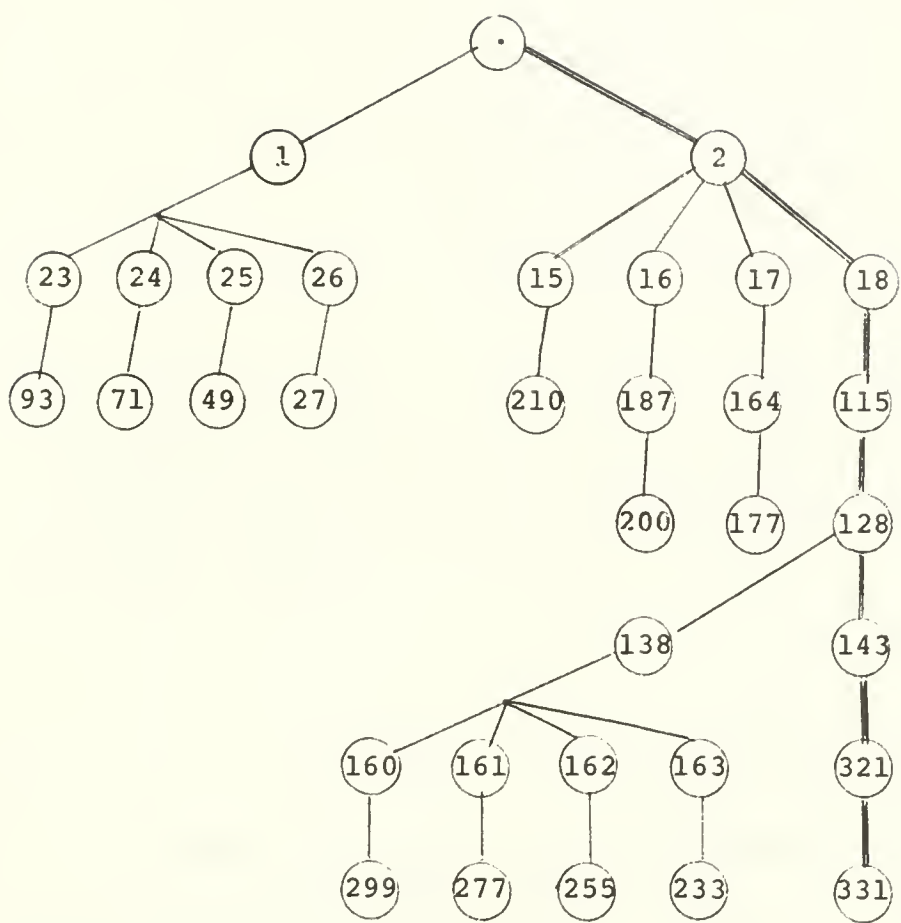
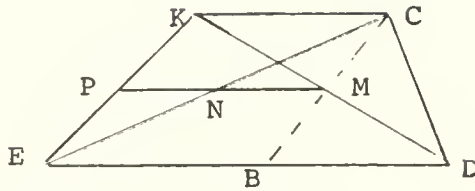


Figure 8. G4 Search Tree. Only nodes
which have complexity 42 or less
are shown.

5.6 The G5 Problem



Given: $EN = NC$, $KM = MD$, $KC \parallel ED$

Prove: $EP = PK$

The G5 problem occurs in two variations. In the first (G5.0) we specify the point B as collinear to ED and that $BM = MC$. In the second version (G5.1) we give no information about B.

G5.0 was solved with the following inputs:

8 coercions as before,

4 special hypotheses and 10 additional clauses:

PEND a line is parallel to a subsection

PBASE a line is parallel to another if the other
is parallel to a subsection

PEXT a line is parallel to its extensions

PARTRANS parallelism is transitive

MIDPAR, ALTER1, VERTICAL, LCONG, LPARBASE, CSAA.

The proof required 8 coercions or 11 resolutions and was found in 153 seconds. The search reached 100 clauses in size.

The G5.1 problem has only been solved by Gelernter's extended system. The necessary construction prevented all the other geometry programs which we have described from solving it.

The G5.1 problem required 677 seconds using the extended system. 571 resolvents were produced. Figure 8 gives the final proof tree. The proof used 16 coercions, or 27 resolutions, and 88 clauses were selected.

EP=PK

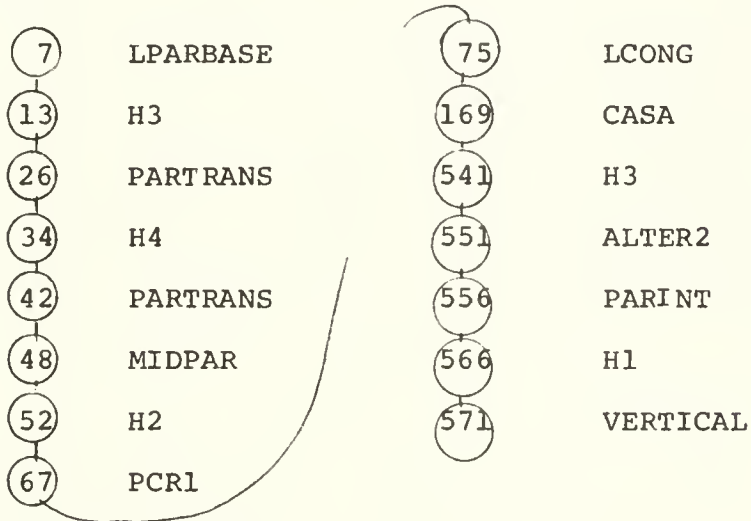


Figure 8. Proof of G5.1

Our results are summarized in Table III.

Problem	Number of Input Clauses		Proof		Final Pool Size
	Coercions	Clauses	Coercions	Resolutions	
G0	8	9	8	23	13
G0.1	8	11	8	23	14
G0.2	8	13	8	23	16
G1	8	12	7	14	58
G	8	12	6	19	7
G3	8	15	8	20	26
G4	8	19	7	20	103
G5.0	8	14	8	11	100
G5.1 *	8	13	16	27	13

* Using extended system.

CHAPTER 6

CONCLUSIONS

6.1 Formalistic Methods

In Chapter 1, we argued that there were two basic approaches to AI -- the formalistic and the intuitive. Perhaps the most basic implication of our research is that formalistic methods may offer a reasonable alternative to intuitive schemes. While we do not claim that formalistic methods are equal in efficiency to intuitive ones, our results show that they are not so slow as to preclude their use. In fact, the added clarity of a formalistic system may make it more useful. We have given a uniform proof procedure and shown that it may be applied to group theory, logical puzzles, and geometry. We believe that any intuitive program which could solve problems in each of these diverse areas would be very awkward.

6.2 Hierarchical Theorem Proving

The figures in Chapter 2 show that on a limited set of theorems, hierarchical theorem proving is more effective than resolution. The basic notion of replacing the requirement of identity with proofs of equivalence inside unification is widely applicable. While we have only used it in Horn clause resolution,

Theorem 5 shows that it could be used in any Herbrand type proof procedure. For example hierarchical notions could be applied in linked conjuncts, model elimination, and all the other unification based schemes.

The coercion mechanism may not be the best way to implement Theorem 5. We expect that difference operations as used in GPS could be significantly more effective. Or one could allow an inner theorem prover, which contained an algebraic simplifier. Certainly the notion of an inner theorem prover could be extended to include additional, even more internal, theorem provers.

6.3 Partial Evaluation and Heuristics

The geometry program shows that partial evaluation is a useful adjunct in a theorem prover. The use of partial evaluation changed a large number of distinct clauses into the same canonical form. It is doubtful if the geometry proofs could have been achieved without such pruning. At the same time, we saw that it was necessary to unevaluate expressions. The hierarchical method allowed just the sort of unevaluation which was needed. We believe that the use of coercion suggests a reasonable way out of the difficulties encountered in QA3 and other partial evaluation systems. Also, the use of predicates to set heuristic switches was

very useful. Such a way of adding heuristics to a uniform proof procedure is natural. The claim that uniform procedures cannot make use of domain dependent information is false.

6.4 The Geometry Program

Finally we come to the most immediate conclusion of this research: that it is possible to use a resolution type theorem prover to solve problems in elementary geometry. When we began this work, such a result would have been considered unlikely. Many AI workers seem to believe that formal theorem provers are unsuitable for such problems.

6.5 Future Possibilities

The completeness theorem given in Chapter 2 states that two level theorem proving may be used with any set of clauses which has a refutation tree. The Dilemma language allows only clauses which are in Horn form. Horn sets are a proper subset of the clauses with a refutation tree, as shown by the following example.

$$S = \left\{ \begin{array}{l} P(a,x) \vee P(b,x) \vee \neg P(c,x) \vee \neg P(d,x) \\ \neg P(a,x) \\ \neg P(b,x) \\ P(c,x) \\ P(d,x) \end{array} \right\}$$

This set is unsatisfiable, has a refutation tree, and cannot be renamed into Horn clause form. We will now describe a splitting algorithm (Chang [5]), which will change an arbitrary set of clauses into several sets of Horn clauses. We will then use S to give an example of the use of this technique.

Chang's Algorithm:

Let $R = R_1 \cup A \vee B$ be a set of clauses.

(1) Prove that $R_1 \cup A$ is unsatisfiable.

If there were n distinct occurrences of instances of A in the refutation let $\sigma_1 \dots \sigma_n$ be mgus such that the i th instance of A is $A\sigma_i$.

(2) R is unsatisfiable if

$R_1 \cup B\sigma_1 \cup B\sigma_2 \dots \cup B\sigma_n$ is unsatisfiable.

In the example, let $A = P(a, x)$

$B = P(b, x) \vee \neg P(c, x) \vee \neg P(d, x)$

$R_1 = S - A \vee B$

Step 1 consists of

$P(a, x)$	
$\neg P(a, x)$	$P(c, x)$
$\neg P(b, x)$	$P(d, x)$

which allows only one resolution, uses only one instance of $P(a, x)$, and involves the null substitution σ .

Step 2 requires proving that $R_1 \cup B\sigma$ is unsatisfiable.

This is a standard Horn clause problem and is easily solved.

This algorithm could be implemented by writing a program which does the splitting and then calls our system in order to solve each subproblem. A new level of hierarchical structure could be provided by modifying our system to produce the refutations of subproblems simultaneously.

Dilemma itself could be extended in several ways. First, we could add a sense of time. Mathematical systems are static; once proved, a theorem stays true. In many situations, however, the truth of a statement changes over time. For example, to a robot a set of theorems could indicate where things are in the current environment. As objects are moved, these theorems would change. We believe that a dynamic component could be added to Dilemma in a way that would make it possible to treat robot control problems conveniently. Another extension would be to add other special inner theorem provers. While the coercion mechanism works well for the equality relation, other relations such as partial ordering or set equality could be added. An open question is whether U-g-resolution can be extended in efficient ways to build in such relations. We expect that additional information about problem solving would be found by writing other programs in Dilemma. For example, a program which does symbolic integration. Here a whole new class of heuristics would be necessary. Possibly an inner theorem prover which knows about algebraic simplification would be required. Finally we could build special hardware.

The Dilemma language is highly parallel. Perhaps a large number of micro processors, each computing a generalized resolution, would lead to a very fast Dilemma machine.

Considering the theorems of Chapter 2, we see that there are a number of unresolved theoretical issues. What happens to completeness when the refutation graph contains cycles? Since these cycles arise from merges, can we restrict how merges occur, for example, to merges on only certain predicate symbols? We have shown that the sets of clauses which are characterized by input proofs, unit proofs, or refutation trees are the same. These are the easy problems for a number of algorithms. Can we generalize this to a complexity measure, i.e. an unsatisfiable set of clauses is of complexity M if the minimum number of merges (i.e. cycles in a refutation graph) is M ? (see Kowalski [23]).

Finally, as we pointed out in Chapter 1, our work is one more step in the direction started by McCarthy toward building programs which would learn. We hope that the Dilemma language and our ideas of hierarchical theorem proving may contribute to the design of an underlying representation of information by a machine. The larger, more global, question raised by our work is how a machine which is given some instruction can transform that instruction into optimal Dilemma procedures.

APPENDIX I

IMPLEMENTATION DETAILS AND LISTING OF THE COMPILER

The implementation is divided into four sections:
basic utilities, input parser, unification, post unification.

Basic Utilities

APPEND(L1,L2)	append LIST L2 to LIST L1 L1 is <u>changed</u> . Result is in the new L1.
RENAME(L1,I)	L1 is a list of literals. I is a number. All variables of L1 are renamed. The left- most variable gets the name I+1.
NAMELIT(lit)	does the renaming. A loop in RENAME calls this procedure.
LOOKUP(item,list)	list is a list of pairs ((x,y)(z,v)). If item is equal to the head of a sublist, the value of lookup is the tail. Bindings are just such lists.
SLB1(C,B)	Apply binding list B to clause C. Change the clause in place.
MAKNAME(cℓ)	Change the variables in a clause to Z1,Z2,...Z12.
DELETE(item,list)	Delete ll copies of item from list. Change list.
MAPR(list,proc)	Like map x but constructs the new list (in place)
PRINTHIST(cℓ)	cℓ clause vector. Print history of the deduction of cℓ.
COPX	A high speed version of the BALM copy.
SUBINPLA(lit,B)	Apply binding list B to literal lit.
SUBER(lit,item,replace)	Change item to replace in literal.

Partial evaluation works inside out (it is the LISP EVAL function). Some tricks are done with the variables DEPTH and MERIT. They look like global variables (but are actually copies).

CLEARSYS drops the BALM system (it gets about 7K additional space)

RTM is the run time monitor; the details of built-in equality are here.

Input Parser

Mostly input is a set of macro transformations. REMCO (remove comma) changes BALM tree structures into Dilemma objects.

Unification

A unifier is a pair (B,L); B = binding list,
L = additional literals (added via coercions)

PUSHUNIT is used to combine two unifiers

MGU1 applies a coercion

COPYPAIR returns the pair of equal or equivalent items associated with a coercion (it only does a copy if the coercion passes the possible test).

Post Unification

If completions are present, clauses retain the resolved literals.

((P 1) (Q 1 2)) resolves with (P (a))

to give

((MARK (P (a)) (Q(a) 2))

MARK is a one argument "predicate" taking the old predicate as its argument. (Many places in the code attempt to bypass marked literals.) If a marked literal is either at the left of a clause or next to another marked literal, its completion is executed and the literal deleted.

*	1
*	2
**	3
*	4
*	5
*	6
**	7
**	8
**	9
DETAIL(X1) MEANS IF DEBUG THEN PRINT(=X1,X1);	10
SETPROP(X1,X2, X3) MEANS SETPROPY(X2,X1,X3);	11
SUBCOPY(X1,X2) MEANS SUBINPLA(COPY(X1), X2);	12
=X1=X2 MEANS BREAKUP(=X1,X2);	13
ATOM(X1) MEANS ¬PAIRQ(X1);	14
FALSE = NIL;	15
**	16
**	17
MACROS TO GET PARTS OF CLAUSES AND LITERALS	18
**	19
EACH CLAUSE IS REPRESENTED BY A 8 ELEMENT ARRAY	20
** 1 CLAUSE ID , *OXX FOR GENERATED NAMES (INPUT CLAUSES) VIA A PROVE	21
NUMBERS FOR RESOLVENTS	22
** 2 PTR TO NEG PARENT	23
** 3 PTR TO LITERALS	24
** 4 LENGTH IN LITERALS	25
ONLY NEGATIVE ATOMS ARE COUNTED	26
** 5 MIXED OR POSITIVE PARENT	27
** 6 EXTRA BINDINGS	28
** 7 DEPTH OF CLAUSE (0 IS AN INPUT CLAUSE)	29
** 8 MERIT OF CLAUSE	30
**	31
NAMEOF(X1) MEANS X1[1] ;	32
PREDSYM(X1) MEANS HD X1;	33
ARGLIST(X1) MEANS TL X1;	34
FNSYM(X1) MEANS PREDSYM(X1);	35
INPUTCL(X1) MEANS IDQ(X1[1]);	36
NEGPARG(X1) MEANS X1[2];	37
LITLIST(X1) MEANS X1[3];	38
CLLENGTH(X1) MEANS X1[4];	39
MIXEDPAR(X1) MEANS X1[5];	40
BINDINGS(X1) MEANS X1[6];	41
DEPTHVAL(X1) MEANS X1[7];	42
MERITVAL(X1) MEANS X1[8];	43
**	44
**	45
GENERAL UTILITY PROGRAMS	46
**	47
COPIES OF BALM UTILITIES WITH CHANGES	48
**	49
** APPEND IS USED TO PUT ONE LIST ON THE BACK OF A SECOND	50
** APPEND WILL CHANGE ITS FIRST ARGUMENT	51
APPEND= PROC(LIST1,LIST2),	52
BEGIN(B),	53
IF ¬LIST1 THEN RETURN LIST2,	54
IF ¬LIST2 THEN RETURN LIST1,	
B= LIST1, WHILE TL LIST1 REPEAT LIST1= TL LIST1,	
TL LIST1= LIST2, RETURN B	

END	55
END;	56
ENAME= PROC(CL, ID),	57
BEGIN(NEWNAMES),	58
NEWNAMES=NIL, RETURN MAPX(CL,	59
NAMELIT)	60
END	61
END;	62
NAMELIT RENAMES THE VARIABLES IN A LITERAL TO NUMBERS THE LEFT MOST	63
VARIABLE GETS THE VALUE OF ID , ID GETS CHANGED	64
AMELIT= PROC(LIT),	65
BEGIN(S),	66
IF ATOM(LIT) THEN DO	67
IF ~(S=LOOKUP(LIT, NEWNAMES)) THEN NEWNAMES= (LIT:(S=ID=ID+1	68
)):NEWNAMES,	69
RETURN S	70
END , TL LIT= MAPX(TL LIT, NAMELIT) , RETURN LIT	71
END	72
END;	73
OOKUP= PROC(ITEM, LIST),	74
BEGIN(),	75
WHILE LIST REPEAT DO	76
IF EQUAL(HD HD LIST, ITEM) THEN RETURN TL HD LIST, LIST= TL	77
LIST	78
END,	79
RETURN NIL	80
END	81
END;	82
UBI= PROC(X, B),	83
BEGIN(Y),	84
* APPLY BINDING LIST B TO X	85
IF ~PAIRQ(X) THEN RETURN SUBINPLA(X, B),	86
Y=X,	87
WHILE X REPEAT DO	88
HD X = SUBINPLA(HD X , B), X= TL X	89
END,	90
RETURN Y	91
END	92
END;	93
AKNAME= PROC(CL),	94
BEGIN(NAMES, XVARS),	95
NAMES=NIL, XVARS=ZVARIABLE, CL	96
=MAPX(CL, NAME1)	97
END	98
END;	99
ELETE= PROC(ITEM, LIST),	100
BEGIN(B, E),	101
B=NIL:LIST,	102
LIST= B,	103
WHILE TL LIST REPEAT DO	104
IF EQUAL(ITEM, HD TL LIST) THEN TL LIST= TL TL LIST ELSE	105
LIST = TL LIST	106
END, RETURN TL B	107
END	108

END;	109
MAPR= PROC(ALIST,PRO),	110
BEGIN(L),	111
L= ALIST, WHILE L REPEAT DO	112
HD L= PRO(HD L), L= TL L	113
END, RETURN ALIST	114
END	115
END;	116
PRINTHIS=PROC(CL),	117
BEGIN(J),	118
J=NIL,	119
IF SAVER THEN DO	120
PRINT(=NULL CLAUSE ***) , RETURN ()	121
END ,	122
WHILE CL REPEAT DO	123
PRINT(NAMEDOF(CL),LITLIST(CL),J), J= MIXEDPAR(CL), CL= NEGP	124
(CL)	125
,J= NAMEDOF(J)	126
END	127
END	128
END;	129
MAX1= PROC(A,B),	130
IF A GT B THEN A ELSE B	131
END;	132
CSTK=MAKVECTO(201);	133
OLDCOPY= COPY;	134
COPX= PROC(Y),	135
BEGIN(I,K,X),	136
IF Y EQ NIL THEN RETURN Y,	137
IF INTQ(Y) THEN RETURN Y,	138
IF IDQ(Y) THEN RETURN Y,	139
IF ~PAIRQ(Y) THEN DO	140
PRINT(=COPY PROBLEMS=, Y),RETURN OLDCOPY(Y)	141
END,	142
IF ~PAIRQ(Y) THEN RETURN Y,	143
K=1,I=3,CSTK[1]=HD Y, CSTK[2]= TL Y,	144
WHILE(K LT I) REPEAT DO	145
IF PAIRQ(X= CSTK[K]) THEN DO	146
CSTK[I]=K,CSTK[I+1]=HD X, CSTK[I+2]= TL X,	147
I= I+3	148
END,	149
IF PAIRQ(X= CSTK[K=(K+1)]) THEN DO	150
CSTK[I]=K,CSTK[I+1]=HD X, CSTK[I+2]=TL X, I= I+3	151
END,	152
IF I GT 198 THEN DO	153
COPYFAIL= COPYFAIL+1,	154
K=X=NIL:NIL,	155
I= Y, WHILE I REPEAT DO	156
X= TL X= COPY(HD I): NIL , I= TL I	157
END,RETURN TL K	158
END,	159
K= K+2	160
END,	161
FOR K=(I-1,5,-3) REPEAT	162

CSTK[CSTK[K-2]] = CSTK[K-1] : CSTK[K],	163
RETURN CSTK[1] : CSTK[2]	164
END	165
END;	166
SUBINPLA = PROC(LIT, UNIF),	167
BEGIN(U, L2),	168
IF ~LIT THEN RETURN NIL,	169
WHILE ~(U.UNIF) = UNIF REPEAT LIT = SUBER(LIT, HD U , TL U)	170
, RETURN (LIT)	171
END	172
END;	173
SUBER = PROC(LIST, ITEM, REPLACE),	174
BEGIN(L2),	175
IF EQUAL(LIST, ITEM) THEN RETURN (REPLACE),	176
IF ATOM(LIST) THEN RETURN LIST,	177
L2 = TL LIST, WHILE L2 REPEAT DO	178
HD L2 = SUBER(HD L2, ITEM , REPLACE), L2 = TL L2	179
END,	180
RETURN (LIST)	181
END	182
END;	183
*	184
* INPUT PARSER	185
*	186
*	187
NAME1 IS A VARIATION OF NAMELIT. HOWEVER, VARIABLES ARE	188
REPLACED BY ELEMENTS OF A LIST RATHER THEN NUMBERS	189
NAME1 = PROC(LIT),	190
BEGIN(S),	191
IF ~LIT THEN RETURN NIL,	192
IF ATOM(LIT) THEN DO	193
S = LOOKUP(LIT, NAMES), IF ~S THEN DO	194
IF ~XVARS THEN DO	195
PRINT(=NAME1, =XVARS, =EMPTY), RETAIN=FALSE, XVARS =	196
XVARIABLE	197
END,	198
S = HD XVARS, XVARS = TL XVARS, NAMES = (LIT:S) : NAMES	199
END,	200
RETURN S	201
END,	202
RETURN HD LIT: MAPX(TL LIT, NAME1)	203
END	204
END;	205
EMB = PROC(ITEM, ALIST),	206
BEGIN(),	207
WHILE PAIRO(ALIST) REPEAT DO	208
IF EQUAL(ITEM, HD ALIST) THEN RETURN TRUE, ALIST = TL ALIST	209
END,	210
RETURN ALIST EQ ITEM	211
END	212
END;	213
EMCO = PROC(X),	214
BEGIN(XX, Y),	215
XX = X,	216

IF ATOM(XX) THEN DO	217
IF MEMBER(XX,CONSTANT	218
) THEN RETURN LIST(XX) ELSE RETURN XX	219
END,	220
IF HD XX EQ = ^ THEN DO	221
ANDSW= TRUE, HD XX= ,	222
END,	223
IF HD XX EQ =, THEN DO	224
IF HD HD TL TL XX EQ = ^ THEN DO	225
ANDSW = TRUE, HD HD TL TL XX = ,	226
END,	227
IF HD HD TL TL XX EQ =, THEN	228
RETURN(REMCO(HD TL XX) : REMCO(HD TL TL XX))	229
ELSE RETURN (REMCO(HD TL XX): REMCO(HD TL TL XX): NIL)	230
END,	231
WHILE XX REPEAT DO	232
IF HD HD XX EQ = ^ THEN DO	233
HD HD XX= , ,ANDSW= TRUE	234
END,	235
IF HD HD XX EQ =, THEN DO	236
Y= HD XX, HD XX= HD TL Y, TL XX=(HD TL TL Y): TL XX	237
END,	238
HD XX = REMCO(HD XX), XX= TL XX	239
END, RETURN X	240
END	241
END;	242
REMCO1= REMCO;	243
CONST = PROC(X),	244
BEGIN(),	245
IF ATOM(X) THEN CONSTANT	246
=X:CONSTANT	247
ELSE	248
CONSTANT	249
=APPEND(REMCO(X),CONSTANT	250
)	251
END	252
END;	253
REMOC1= REMCO;	254
DIFFEREN (X1,X2) MEANS NOTEQLIS= (REMCO(=X1) : REMCO(=X2)): NOTEQLIS;	255
BRACKET(=CONSTANT	256
,100,=CONSTANT	257
);	258
CONSTANT	259
X1	260
END MEANS CONST(=X1);	261
INFIX(=ASSERT,1451,1450,=ASSERT);	262
X1 ASSERT X2 MEANS MIXEDCL=GENCLAUS(=X1,REMOC1(=X2),NIL):MIXEDCL;	263
INFIX(=THEOREM,1501,1500,=THM);	264
INFIX(=FROM,1400,1400,=FRM);	265
INFIX(=COER,1451,1450,=COER);	266
UNARY(=TYPE,100,=TY);	267
TYPE X1 = X2 MEANS SETPROP(=X1,=TYPE ,X2);	268
FIXCO= PROC(X),	269
BEGIN(Y),	270

ANDSW= FALSE,	271
Y=REMCQ(X), IF ATOM(Y) THEN RETURN Y,	272
IF ~ANDSW THEN RETURN LIST(Y),	273
RETURN Y	274
END	275
END;	276
INFIX(= ^ , 1401,1401, = ^);	277
X1 THEOREM X2 FROM X3 MEANS MIXEDCL=GENCLAUS(=X1,REMCQ(=X2),FIXCO	278
(=X3)): MIXEDCL ;	279
X1 COER X2, X3 MEANS GENCOER(=X1,REMCQ1(=X2),REMCQ1(=X3),NIL);	280
X1 COER X2,X3,X4 MEANS GENCOER(=X1,REMCQ1(=X2),REMCQ1(=X3),FIXCO(=X4));	281
PROVE(X1) MEANS OTTER(=X1);	282
INFIX(=COMPLETE,1401,1400,=CPL);	283
INFIX(=CPROC,1401,1400,=CPRPC);	284
INFIX(=PART,1401,1400,=PART);	285
INFIX(=PARTEVAL,1401,1400,=PE);	286
X1 COMPLETE X2 MEANS DO	287
SETPROP(=X2,=CMPL,X1), COMPLET = TRUE	288
END;	289
X1 CPROC X2 MEANS DO	290
SETPROP(=X2,=CPROC,X1)	291
END;	292
X1 PART 1 = X2 MEANS DO	293
PEFNS= TRUE, SETPROP(=X1,=PE1,X2)	294
END;	295
X1 PART 2 = X2 MEANS DO	296
PEFNS= TRUE, SETPROP(=X1,=PE2,X2)	297
END;	298
X1 PART 3 = X2 MEANS DO	299
PEFNS= TRUE, SETPROP(=X1,=PE3,X)	300
END;	301
X1 PART 4 = X2 MEANS DO	302
PEFNS= TRUE, SETPROP(=X1,=PE4,X2)	303
END;	304
X1 PART 5 = X2 MEANS DO	305
PEFNS= TRUE, SETPROP(=X1,=PE5,X2)	306
END;	307
X1 PART 6 = X2 MEANS DO	308
PEFNS= TRUE, SETPROP(=X1,=PE6,X2)	309
END;	310
X1 PARTEVAL X2 MEANS DO	311
PEFNS= TRUE, SETPROP(=X1,=PE,X2)	312
END;	313
*	314
* CODE GENERATORS	315
*	316
* GENCLAUSE GETS NAME OF CLAUSE, POSITIVE LITERAL, AND A LIST OF	317
* NEGATIVE LITERALS AS ITS THREE ARGUMENTS.	318
* IT CREATES A VECTOR RESULT	319
* NOTE THAT PE DONT GET EXECUTED HERE NOR DO COMPLETIONS OF POSITIVE	320
* UNITS	321
* GENCLAUSE DOES NOT RENAME CLAUSES TO XVARS, NOR DOES IT CHECK	322
* IF A CLAUSE SUBSUMES OR IS A DUPLICATE OF AN EXISTING CLAUSE	323
* PROC GENNEG IS THE RUNTIME GENERATOR . IT IS IN THE POST UNIFICATION	324

* SECTION.	325
GENCOER= PROC(NAME,A,B,C),	326
BEGIN(XVARS, NAMES),	327
NAMES= NIL,XVARS=ZVARIABLE,	328
COERS= NAME:COERS,	329
SETPROP(NAME,=PAIR,NAME1(A):NAME1(B))	330
, VALUE NAME= VECTOR(NAME,NIL,MAPX(C,NAME1),LENGTH(C),NIL,NIL,0	331
,0),	332
V= COPY(ZVARIABLE), WHILE XVARS REPEAT DO	333
V= DELETE(HD XVARS,V), XVARS= TL XVARS	334
END,	335
DETAIL(V),	336
SETPROP(NAME,=VARS,V), RETURN VALUE NAME	337
END	338
END;	339
GENCLAUS=PROC(NAME,POS,NEG),	340
BEGIN(P),	341
VALUE NAME = VECTOR(NAME,NIL,NIL,LENGTH(NEG) ,NIL,NIL,0,0),	342
LITLIST(VALUE NAME) = NEG,	343
SETPROP(NAME,=POSLIT,POS),	344
MERITVAL(VALUE NAME)= COMPLEX(NEG) ,	345
IF POS THEN DO	346
P= PREDCHAI, WHILE P REPEAT DO	347
IF HD POS EQ HD HD P THEN DO	348
TL HD P=(VALUE NAME):(TL HD P),	349
RETURN VALUE NAME	350
END, P= TL P	351
END,	352
PREDCHAI= ((HD POS):((VALUE NAME):NIL)):PREDCHAI	353
END,	354
RETURN VALUE NAME	355
END	356
END;	357
* UNIFICATION	358
* UNIFICATION	359
* UNIFICATION	360
ZVARIABLE= (Z1 Z2 Z3 Z4 Z5 Z6 Z7 Z8 Z9 Z10 Z11 Z12);	361
SETBIND(X1,X2) MEANS	362
(((X1:X2):NIL):NIL) ;	363
INFIX(=ISNOTIN,1401,1200,=NOTIN);	364
EMPTY= LIST(NIL);	365
EMPTY = LIST(LIST(NIL));	366
** NOTIN IS A UNIFICATION PRIMITIVE VAR NOTIN EXP	367
NOTIN= PROC(VAR,ALIST),	368
BEGIN(),	369
IF ~PAIRQ(ALIST) THEN RETURN (~ EQUAL(VAR,ALIST)),	370
ALIST= TL ALIST,	371
WHILE ALIST REPEAT DO	372
IF ~NOTIN(VAR, HD ALIST) THEN RETURN FALSE,	373
ALIST= TL ALIST	374
END , RETURN TRUE	375
END	376
END;	377
* THE PAIR OF A COERCION ARE COPIES OF THE TWO EQUIV EXPRESSIONS	378

* WHERE VARIABLES HAVE BEEN REPLACED BY NUMBERS	379
PUSHUNIF=PROC(OLD,NEW),	380
BEGIN(U1,U2,L1,L2,U3,U,P,L4),	381
** OLD AND NEW ARE TWO UNIFER LISTS WHICH ARE TO BE COMBINED AND UPDATE	382
IF EQUAL (OLD , EMP1) THEN RETURN NEW,	383
IF EQUAL(EMP1,NEW) THEN RETURN OLD,	384
=(U1.L1)=OLD,=(U2.L2)=NEW,	385
L1= SUB1(L1,U1), L1= SUB1(L1,U2),	386
L2= SUB1(L2,U2),	387
L4=L2,	388
IF L2 THEN	389
DO	390
WHILE TL L2 REPEAT L2= TL L2, TL L2= L1	391
END ELSE L4= L1,	392
U3 = U1,	393
WHILE =(P.U3)=U3 REPEAT TL P= SUBINPLA(TL P,U2),	394
U3 = U1,	395
IF U3 THEN DO	396
WHILE TL U3 REPEAT U3= TL U3, TL U3=U2	397
END ELSE U1=U2,	398
RETURN U1:L4	399
END	400
END;	401
GUI= PROC(P,N,A,B,OUT,DEPTH,COR,BINDS,NAME),	402
BEGIN(M1,M,P1,U,M2,M3, Mhold),	403
M=M2=NIL,	404
IF BVARs THEN DO	405
IF ATOM(P) AND ATOM(N) THEN RETURN UOUT	406
END,	407
UNIFDEPT=UNIFDEPT+1,	408
IF BVARs THEN DO	409
IF ATOM(P) AND ATOM(A) THEN M= LIST(LIST(A:P):NIL)	410
END , IF ~M THEN	411
M= (IF U1 THEN EMPTY ELSE MGOSET(P,A,DEPTH,ADDLITS- LENGTH(COR)	412
,NAME)),	413
WHILE =(M1.M) = M REPEAT DO	414
M2= NIL,	415
IF BVARs THEN DO	416
IF ATOM(N) AND ATOM(B) THEN M2= LIST(LIST(B:N):NIL)	417
END ,IF ~M2 THEN	418
M2= (IF U2 THEN EMPTY ELSE MGOSET(SUBCOPY(N,HD M1),SUBCOPY(B	419
,HD M1),	420
DEPTH,ADDLITS- LENGTH(COR)- LENGTH(TL M1),NAME)),	421
WHILE(=(M3.M2)=M2) REPEAT DO	422
UOUT= PUSHUNIF(PUSHUNIF(COPY(M1),M3),(COPY(BINDS):COPY(COR))):UOUT	423
END	424
END,	425
UNIFDEPT=UNIFDEPT-1,	426
RETURN UOUT	427
END	428
END;	429
SSIBLE=PROC(A, N , FLAG),	430
BEGIN(),	431
	432

VALUE FLAG = NIL,	433
IF ATOM(A) THEN RETURN TRUE,	434
IF ATOM(N) THEN RETURN TRUE ,	435
IF HD A EQ HD N THEN RETURN TRUE, RETURN FALSE	436
END	437
END;	438
MGUSET= PROC(N,P,DEPTH,ADDLITS,DONT),	439
BEGIN(NT,PT,P1,N1, UNIFS, UOUT,E1, U2,B, U1,A,E,M, LCOR, BIND,	440
BINDS,	441
U,J,J2,NAME),	442
IF EQUAL(N,P) THEN RETURN (EMPTY),	443
N1= NOTEQLIS,	444
WHILE N1 REPEAT DO	445
NT= HD N1, N1= TL N1,	446
IF EQUAL(HD NT,P) AND EQUAL(TL NT,N) THEN RETURN =NOGOOD,	447
IF EQUAL(HD NT,N) AND EQUAL(TL NT,P) THEN RETURN =NOGOOD	448
END,	449
IF DEPTH GT MAXCOERS THEN RETURN =NOGOOD,	450
UOUT=UNIFS=NIL,	451
IF ~BYPASS THEN DO	452
IF ATOM(P) THEN DO	453
IF P ISNOTIN N THEN UNIFS= LIST(SETBIND(P,N))	454
END	455
ELSEIF ATOM(N) THEN DO	456
IF N ISNOTIN P THEN UNIFS= LIST(SETBIND(N,P))	457
END,	458
UOUT=UNIFS,	459
IF PAIRQ(N) AND PAIRQ(P) AND(HD N EQ HD P) THEN DO	460
UNIFS=EMPTY,	461
NT = TL N, PT= TL P,	462
WHILE NT REPEAT DO	463
N1= HD NT, NT= TL NT,	464
P1= HD PT, PT= TL PT, UOUT= NIL,	465
WHILE UNIFS REPEAT DO	466
U= HD UNIFS, UNIFS= TL UNIFS,	467
IF ADDLITS LT LENGTH(TL U) THEN M= =NOGOOD ELSE DO	468
A= COPY(N1),B=COPY(P1),	469
UNIFDEPT=UNIFDEPT+1,	470
M=MGUSET(SUBINPLA(A,HD U), SUBINPLA(B,HD U),	471
DEPTH,	472
ADDLITS-LENGTH(TL U),NIL),	473
UNIFDEPT=UNIFDEPT-1	474
END,	475
WHILE PAIRQ(M) REPEAT DO	476
E= HD M, M= TL M,	477
UOUT=PUSHUNIF(COPY(U),E):UOUT	478
END	479
END, UNIFS= UOUT	480
END	481
END, BYPASS=NIL,	482
E1=(ATOM(N) OR ATOM(P)),	483
IF ~E1 THEN E1= (HD N EQ HD P),	484
IF ~E1 THEN E1= ALWAYS,	485
	486

IF E1 THEN DO	487
IF (DEPTH LT MAXCOERS) THEN DO	488
E= COERS,	489
WHILE E REPEAT DO	490
E1= HD E, E= TL E,	491
IF E1 NE DONT THEN DO	492
NAME= E1,	493
IF CLENGTH(VALUE E1) LE ADDLITS THEN DO	494
J2=GETPROP(E1,=CPROC),	495
IF COPYPAIR(E1,N,P) THEN DO	496
M= BINDS,	497
IF(¬ATOM(A) AND ¬ATOM(B)) OR (UNIFDEPT NE 0)	498
THEN DO	499
BINDS= NIL,	500
E1= LCOR,	501
IF CODEQ(J2) THEN J=J2(N,P,UOJT) ELSE J=	502
TRUE,	503
IF J THEN DO	504
N1= 0, IF J NE TRUE THEN N1=(MAXCOERS -	505
J), N1= MAX1(N1,DEPTH+1),	506
IF POSSIBLE(A,N,=U1) AND POSSIBLE(B,P,=	507
U2) THEN DO	508
UOUT=MGU1(N,P,A,B,OUT,N1,COPY(E1),	509
BINDS,NAME)	510
END	511
END,	512
IF (¬ONEWAY) OR (¬UOUT) THEN DO	513
IF CODEQ(J2) THEN J=J2(P,N,UOUT) ELSE J	514
= TRUE,	515
IF J THEN DO	516
N1= 0, IF J NE TRUE THEN N1=(517
MAXCOERS -J), N1= MAX1(N1,DEPTH+1),	518
IF POSSIBLE(B,N,=U2) AND POSSIBLE(A,	519
P,=U1) THEN DO	520
UOUT=MGU1(P,N,A,B,OUT,N1,E1,BINDS	521
,NAME)	522
END	523
END	524
END	525
END	526
END	527
END	528
END	529
END	530
END,	531
IF ¬UOUT THEN UOUT = =NOGOOD,	532
RETURN UOUT	533
END	534
END ;	535
COPYPAIR=PROC(COR,S,T),	536
BEGIN(P,V,V1),	537
P=GETPROP(COR,=PAIR),	538
IF¬POSSIBLE(HD P,S,=V) THEN DO	539
	540

IF ¬POSSIBLE(TL P, S, =V) THEN RETURN FALSE,	541
IF ¬POSSIBLE(HD P, T, =V) THEN RETURN FALSE	542
END,	543
IF ¬POSSIBLE(TL P, T, =V) THEN RETURN FALSE,	544
P= COPY(GETPROP(COR,=PAIR)),V= GETPROP(COR,=VARS),	545
BIND= NIL, WHILE =(V1.V) = V REPEAT	546
BIND=(V1:(CORID=CORID+1)):BIND, LCOR= SUB1(COPY(LITLIST(VALUE	547
COR))	548
, BIND),	549
A=SUBINPLA(COPY(HD P),BIND),B=SUBINPLA(COPY(TL P),BIND),	550
RETURN TRUE	551
END	552
END;	553
VARSNUM= PROC(LIT),	554
BEGIN(S,L2),	555
S=0,L2=TL LIT, WHILE L2 REPEAT DO	556
IF ATOM(HD L2) THEN S= S+1 ELSE S= S+VARSNUM(HD L2),L2= TL	557
L2	558
END,	559
RETURN S	560
END	561
END;	562
*	563
* POST UNIFICATION PROCESSING	564
*	565
EMPTYCLA= PROC(X),	566
CLENGTH(X) EQ 0	567
END;	568
* SUBTRACT IS USED TO BUILD RESOLVENTS.	569
* IT BOTH APPLIES A UNIFIER AND REMOVES OCCURANCES OF THE LITERAL	570
SUBTRACT=PROC(LIST,LIT,UNIF),	571
BEGIN(L, L1),	572
LIT= SUBINPLA(LIT,UNIF),	573
L= NIL,	574
WHILE LIST REPEAT DO	575
L1= SUBCOPY(HD LIST,UNIF),	576
IF ¬EQUAL(L1,LIT) THEN L= L1:L,	577
LIST= TL LIST	578
END, RETURN L	579
END	580
END;	581
FAKECL=VECTOR(=FAKECL,NIL,0,NIL,NIL,0,-100);	582
Cplex= PROC(CL),	583
MERITVAL(CL) + DEPTHVAL(CL) / MAXCplex	584
END;	585
NEST= PROC(CL,D),	586
BEGIN(M,M2),	587
IF ATOM(CL) THEN RETURN D,	588
M=0,	589
WHILE CL REPEAT DO	590
M2= NEST(HD CL, D+1),	591
IF M2 GT M THEN M= M2, CL= TL CL	592
END,	593
RETURN M	594

END	595
END;	596
CLMEMBER=PROC(CL,LIST),	597
BEGIN(M,w),	598
M=MERITVAL(CL),	599
CL=LITLIST(CL),	600
WHILE LIST REPEAT DO	601
W= HD LIST, IF MERITVAL(W) EQ M THEN DO	602
IF SAMECL(CL,W) THEN DO	603
IF CLAUSEPR THEN	604
PRINT(*DUPLICATE OF*,NAMEOF(W)), RETURN TRUE	605
END	606
END,	607
LIST= TL LIST	608
END, RETURN NIL	609
END	610
END;	611
CLEC= PROC(LIT1,LIT2),	612
BEGIN(),	613
IF EQUAL(LIT1,LIT2) THEN RETURN TRUE,	614
IF ((PREDSYM(LIT1) EQ EQ1) OR (PREDSYM(LIT1) EQ EQ2))	615
AND ((PREDSYM(LIT2) EQ EQ1) OR (PREDSYM(LIT2) EQ EQ2))	616
THEN	617
DO	618
IF EQUAL(TL LIT1,TL LIT2) THEN RETURN TRUE,	619
IF ~ EQUAL(HD TL LIT1,HD TL TL LIT2) THEN RETURN FALSE,	620
RETURN EQUAL(HD TL LIT2, HD TL TL LIT1)	621
END, RETURN FALSE	622
END	623
END;	624
SAMECL= PROC(LIT,CPR),	625
BEGIN(CPRLIT),	626
CPRLIT=LITLIST(CPR),	627
RETURN SEQUAL(LIT,CPRLIT)	628
END	629
END;	630
MOVE= PROC(LIST),	631
BEGIN(),	632
WHILE LIST REPEAT DO	633
IF HD HD LIST NE =MARK THEN RETURN LIST, LIST= TL LIST	634
END, RETURN NIL	635
END	636
END;	637
GENNEG= PROC(NCTX,MCTX,NEGLIT,UNIFER,NEGLITS),	638
BEGIN(CL,PLIT,LITS),	639
NEGLITS= COPY(NEGLITS),	640
PLIT=COPY(LITLIST(MCTX)),	641
CURCONTE = CL= VECTOR (CLAUSEID=CLAUSEID+1,NCTX,NIL,LENGTH(PLIT	642
)	643
+ CLENGTH(NCTX) -1, MCTX,NIL,DEPTHVAL(NCTX)+1,MERITVAL(NCTX)),	644
IF SAVER THEN MIXEDPAR(CURCONTE)= NIL,	645
IF SAVER THEN NEGPAR(CURCONTE)= NIL,	646
LITS= APPEND(TL UNIFER,APPEND(PLIT,NEGLITS)),	647
DETAIL(LITS),	648

PLIT=LITS, WHILE PLIT REPEAT DO	649
HD PLIT = SUBINPLA(HD PLIT,HD UNIFER), PLIT = TL PLIT	650
END,	651
DETAIL(LITS),	652
IF COMPLET THEN	653
LITS=MARK(LITS,SUBINPLA(NEGLIT,HD UNIFER)) ELSE LITS= DELETE(654
SUBINPLA(NEGLIT,HD UNIFER),LITS),	655
DETAIL(LITS),	656
MERITVAL(CL)= COMPLEX(LITS),	657
IF PEFNS THEN DO	658
NEWBINDS= NIL, NAMES=NIL,XVARS=XVARIABLE,LITS=MAPX(LITS,NAME1	659
),	660
IF RETAIN THEN	661
LITS= EVALUATE(LITS)	662
END,	663
DETAIL(LITS),	664
IF COMPLET THEN	665
TESTCOMP(LITS),	666
IF COMPLET THEN	667
LITS= MERGE(LITS),	668
LITS= RENAME(RENAME(LITS,CORID+2),0),	669
DETAIL(LITS),	670
LITLIST(CL)=LITS,	671
CLENGTH(CL)= LENGTH(LITS),	672
RETURN CL	673
END	674
END;	675
SEQUAL= PROC(LITS,LITS2),	676
BEGIN(),	677
LITS=MOVE(LITS), LITS2=MOVE(LITS2),	678
WHILE LITS REPEAT DO	679
IF ~CLEQ(HD LITS , HD LITS2) THEN RETURN FALSE,	680
LITS=MOVE(TL LITS),LITS2=MOVE(TL LITS2),	681
IF ~ LITS THEN RETURN ~LITS2	682
END, RETURN FALSE	683
END	684
END;	685
ADDNEGS= PROC(CL),	686
BEGIN(N,M,B),	687
N=FAKECL:NEGS,M=CPLEX(CL),B=N,	688
RACCPT= RACCPT+1,	689
NUMSOLUT= NUMSOLUT+1,	690
WHILE TL N AND(CPLEX(HD TL N) LT M)REPEAT	691
N= TL N, TL N= CL: TL N, NEGS= TL B, RETURN()	692
END	693
END;	694
DELATE= PROC(CL,RETAIN),	695
BEGIN(),	696
IF DEPTHVAL(CL) GT MAXLEVEL THEN DO	697
RETAIN= FALSE,	698
DEEPCNTR= DEEPCNTR+1,	699
PRINT(*DEPTH CUTOFF*)	700
END	701
ELSE IF NEST(LITLIST(CL),0) GT MAXNEST THEN DO	702

NSTR= NSTR+1,	703
PRINT(*TOO DEEP*), RETAIN= FALSE	704
END	705
ELSE IF MERITVAL(CL) GT MAXCPX THEN DO	706
CNTR= CNTR+1,	707
RETAIN = FALSE, PRINT(*TOO COMPLEX*)	708
END,	709
RETURN RETAIN	710
END	711
END;	712
* PARTIAL EVALUATION OF PREDICATES AND FUCTIONS	713
*	714
*	715
CADJUST(X1) MEANS IF MEMBER(X1,CONSTANT	716
) THEN X1= LIST(X1);	717
PENAMES = =(PE1 PE2 PE3 PE4 PE5 PE6);	718
* WHEN WE START TO PE A CLAUSE WE FIRST CHANGE ALL VARIABLES INTO	719
* MEMBERS OF THE LIST VARIABLES. THIS LETS IDQ WORK	720
* ISBOUND RETURNS TRUE IS ITS ARGUMENT HAS A BINDING ON THE SPECIAL	721
* BINDING LIST FOR THIS CLAUSE	722
XVARIABLE= =(X1 X2 X3 X4 X5 X6 X7 X8 X9 X10 X11 X12);	723
ISBOUND= PROC(VAR,CONTEXT),	724
BEGIN(),	725
CADJUST(VAR),	726
IF(BNDVAL=LOOKUP(VAR,NEWBINDS))THEN RETURN TRUE,	727
IF NUMARGS() EQ 1 THEN CONTEXT= CURCONTE,	728
RETURN ISBND1(VAR,CONTEXT)	729
END	730
END;	731
ISBND1= PROC(VAR,CTX),	732
BEGIN(),	733
* THE GLOBAL VARIABLE BNDVAL GETS THE BINDING	734
IF (BNDVAL= LOOKUP(VAR,BINDINGS(CTX))) THEN RETURN TRUE,	735
IF INPUTCL(CTX) THEN RETURN FALSE,	736
IF ISBND1(VAR,NEGPARG(CTX)) THEN RETURN TRUE	737
END	738
END;	739
VARIABLE=PROC(ITEM),	740
¬PAIRQ(ITEM) AND ¬MEMBER(ITEM,CONSTANT	741
)	742
END;	743
BIND= PROC(EXP1,EXP2),	744
BEGIN(J),	745
CADJUST(EXP1),CADJUST(EXP2),	746
IF ¬SAVER THEN	747
NEWBINDS=(EXP1 : EXP2) :NEWBINDS,	748
MASTPRED=(SUBINPLA(MASTPRED,EXP1: EXP2 :NIL))	749
END	750
END;	751
PE= PROC(PRED),	752
BEGIN(S,A,J),	753
S=PREDSYM(PRED), A=ARGLIST(PRED), MAPR(A,PEVAL1),	754
IF CODEQ(J=GETPROP(S,=PE)) THEN RETURN J(PRED)	755
ELSE RETURN PRED	756

END	757
END;	758
PEVAL= PROC(FN),	759
BEGIN(S,A,J),	760
S=FNSYM(FN),A=ARGLIST(FN), MAPR(A,PEVAL),	761
IF CODEQ(J=GETPROP(S,=PE))THEN DO	762
S=J(FN),	763
CADJUST(S), RETURN S	764
END,	765
RETURN FN	766
END	767
END;	768
PEVAL1= PROC(X),	769
BEGIN(Y),	770
IF ATOM(X) THEN RETURN X, IF ~TL X THEN RETURN X ,	771
Y=PEVAL(COPY(X)),	772
IF ~EQUAL(X,Y) THEN BIND(X,Y) , RETURN Y	773
END	774
END;	775
EVALUATE = PROC(CL),	776
BEGIN(B,NEWBINDS),	777
B= CL,	778
WHILE B REPEAT DO	779
NEWBINDS=NIL,	780
IF HD HD B NE =MARK THEN DO	781
WORK=PARTEV(HD B),	782
DETAIL(WORK),	783
IF WORK EQ TRUE THEN CL= DELETE(HD B, CL),	784
IF WORK EQ FALSE THEN DO	785
RETAIN= FALSE, PRINT(≠EVAL REJECTS≠),	786
RETURN CL	787
END,	788
IF NEWBINDS THEN DO	789
S= NIL, SUBINPLA(NIL:CL, NEWBINDS),	790
WHILE NEWBINDS REPEAT DO	791
IF ATOM(HD HD NEWBINDS) THEN DO	792
IF ~ MEMBER(HD HD NEWBINDS, XVARIABLE) THEN DO	793
S= (HD NEWBINDS) : S	794
END	795
END,NEWBINDS= TL NEWBINDS	796
END,	797
WHILE S REPEAT DO	798
NEWBINDS= (HD S): NEWBINDS, S= TL S	799
END,	800
IF NEWBINDS THEN BINDINGS(CURCONTE) = NEWBINDS:	801
BINDINGS(CURCONTE)	802
END	803
END,B= TL B	804
END, RETURN CL	805
END	806
END;	807
CRACK= PROC(PAT,EXP),	808
BEGIN(S),	809
IF IDQ(PAT) THEN DO	810

IF(CODEQ(S=GETPROP(PAT,=TYPE))) THEN DO	811
IF PAIRQ(EXP) AND ~ARGLIST(EXP) THEN DO	812
S= S(HD EXP),EXP= HD EXP	813
END	814
ELSE S= S(EXP),	815
IF ~ S THEN RETURN FALSE	816
END,	817
VALUE PAT = EXP, RETURN TRUE	818
END,	819
IF ATOM(PAT) THEN RETURN EQUAL(PAT,EXP),	820
IF ATCM(EXP) THEN RETURN FALSE,	821
IF CRACK(HD PAT,HD EXP) THEN RETURN CRACK(TL PAT, TL EXP),	822
RETURN FALSE	823
END	824
END;	825
PARTEV = PROC(PRED),	826
BEGIN(CONTEXT,MERIT,S,NFN,SA,DEPTH,J,V,MASTPRED),	827
MASTPRED=PRED,	828
CONTEXT=CURCONTE,MERIT=MERITVAL(CONTEXT), DEPTH=DEPTHVAL(829
CONTEXT),	830
S= PE(PRED),	831
MERITVAL(CONTEXT)=MERIT,	832
IF ~PAIRQ(S) THEN RETURN S,	833
IF ~PAIRQ(HD S) THEN RETURN S ,	834
NFN= PENAMES, SA= ARGLIST(S),	835
WHILE SA REPEAT DO	836
IF VARIABLE(HD SA) THEN DO	837
IF CODEQ(J=GETPROP(PREDSYM(S),HD NFN)) THEN DO	838
V= J(PRED),	839
IF V NE PRED THEN DO	840
BIND(HD SA,V), RETURN TRUE	841
END	842
END	843
END,	844
NFN=TL NFN,SA= TL SA	845
END,	846
MERITVAL(CONTEXT)=MERIT, RETURN S	847
END	848
END;	849
GROUND= PROC(X),	850
BEGIN(T),	851
IF ATCM(X) THEN DO	852
(IF VARIABLE(X) THEN RETURN FALSE),	853
RETURN TRUE	854
END,	855
T= TL X,	856
WHILE T REPEAT DO	857
IF ~GROUND(HD T) THEN RETURN FALSE, T= TL T	858
END,	859
RETURN TRUE	860
END	861
END;	862
HASCON= PROC(LIT),	863
BEGIN(),	864

IF ATOM(LIT) THEN RETURN FALSE,	865
IF ~ TL LIT THEN RETURN TRUE,	866
LIT= TL LIT,	867
WHILE LIT REPEAT DO	868
IF HASCON(HD LIT) THEN RETURN TRUE,LIT=TL LIT	869
END,	870
RETURN FALSE	871
END	872
END;	873
FREEVARS= PROC(LITS),	874
BEGIN(L,H,S),	875
S=0,L=NIL,H=LITS,	876
WHILE H REPEAT DO	877
GETFREE(HD H), H= TL H	878
END, RETURN S	879
END	880
END;	881
GETFREE= PROC(LIT),	882
BEGIN(T),	883
IF ATOM(LIT) THEN DO	884
IF ~L THEN DO	885
L= LIT:NIL,S=1,RETURN NIL	886
END,	887
IF MEMBER(LIT ,L) THEN RETURN NIL,	888
L=LIT:L,S=S+1,RETURN NIL	889
END,	890
T= TL LIT,	891
WHILE T REPEAT DO	892
GETFREE(HD T), T= TL T	893
END, RETURN NIL	894
END	895
END ;	896
*	897
*	898
*	899
ADD MIXED=PROC(NEWCL),	900
MIXEDCL=NEWCL:MIXEOCL	901
END;	902
* CL IS A LIST OF LITERALS	903
* WE REMOVE EACH LITERAL EQUAL TO LIT AND REPLACE IT BY A	904
* ONE ARGUMENT PREDICATE =MARK THE LITERAL BECOMES THE ARGUMENT	905
* LATER THE CLAUSE WILL BE CHECKED FOR COMPLETIONS	906
* IF A MARKED LITERAL IS AT THE TOP OR TWO MARKED LITERALS ARE	907
* TOGETHER THEN THEY MAY BE COMPLETED	908
MARK= PROC(CL,LIT),	909
BEGIN(B),	910
B= CL,	911
WHILE CL REPEAT DO	912
IF EQUAL(HD CL,LIT) THEN HD CL= LIST(=MARK,LIT),	913
CL= TL CL	914
END , RETURN B	915
END	916
END;	917
* MERGE REMOVES MARKED LITERALS FROM THE TOP OF A LIST	918

* IT DOES NOT CHECK FOR MERGES	919
* HOWEVER THIS IS JUST THE PLACE TO PUT IN THAT CHECK	920
MERGE= PROC(CL),	921
BEGIN(),	922
WHILE HD HD CL EQ =MARK REPEAT CL = TL CL, RETURN CL	923
END	924
END;	925
* MOVE FINDS THE FIRST NON MARKED LITERAL OF A LIST	926
MOVE= PROC(LIST),	927
BEGIN(),	928
WHILE LIST REPEAT DO	929
IF HD HD LIST NE =MARK THEN RETURN LIST,	930
LIST= TL LIST	931
END, RETURN NIL	932
END	933
END;	934
** TEST FOR COMPLETIONS MARKED LITS AT TOP OR NEXT TO EACHOTHER	935
TESTCOMP=PROC(CL),	936
BEGIN(SWITCH),	937
SWITCH= TRUE,	938
WHILE CL REPEAT DO	939
IF HD HD CL EQ =MARK THEN	940
(IF SWITCH THEN COMPLETR(HD CL) ELSE SWITCH = TRUE)	941
ELSE SWITCH = FALSE, CL= TL CL	942
END	943
END	944
END;	945
COMPLETR=PROC(LIT),	946
BEGIN(S,J,TEMP,G,T2,L,N,NEGLITS,UN,U,CL),	947
S=PREDSYM(LIT),	948
IF(~(J=CODEQ(GETPROP(S,=CMPLT)))) THEN RETURN NIL,	949
TEMP=J(LIT),	950
WHILE TEMP REPEAT DO	951
IF HD TEMP EQ TRUE THEN DO	952
G= GENCLAUS(GENSYM(),NIL,LIST(LIT)),	953
ADMIXED(G)	954
END	955
ELSE DO	956
T2= GETPROP(HD TEMP,=DESEND): (HD TEMP),	957
WHILE(=(CL.T2)= T2) REPEAT DO	958
NEGLITS=LITLIST(CL),	959
WHILE =(N.NEGLIT)=NEGLITS REPEAT DO	960
UN= MGUSET(N,COPY(LIT), MAXCOERS+1,0),	961
WHILE=(U.UN)=UN REPEAT DO	962
ADMIXED(GENCLAUS(G=GENSYM(),SUBINPLA(POSLIT OF	963
CL,HD U)),	964
SUBTRACT(LITLIST(CL), LIT,HD U))),	965
IF CLENGTH(G) EQ 0 THEN COMPLETR (G)	966
END	967
END	968
END	969
END,	970
TEMP= TL TEMP	971
END,RETURN TRUE	972

END	973
END;	974
**	975
**	976
**	977
PRINTS= PROC(),	978
BEGIN(),	979
S= COERS, PRINT(*COERCIONS*),	980
WHILE S REPEAT DO	981
J= VALUE (HD S),	982
PRINT(HD S, LITLIST(J), GETPROP(HD S, *PAIR)), S= TL S	983
END,	984
S= MIXEDCL, PRINT(*MIXEDCL + POSITIVE CLAUSES*),	985
WHILE S REPEAT DO	986
PRINT(J= NAMEOF(HD S), GETPROP(J, *POSLIT), *FROM*, LITLIST(HD S)), S= TL S	987
END,	988
RETURN TRUE	989
END	990
END;	991
OTTER= PROC(X),	992
BEGIN(S, STIME),	993
ANDSW= FALSE,	994
PRINTS(),	995
X= REMCO(X),	996
IF ATOM(HD X) THEN S= LIST(LIST(X))	997
ELSE IF ATOM(HD HD X) THEN DO	998
IF ANDSW THEN S= LIST(X) ELSE DO	999
S= X, WHILE S REPEAT DO	1000
HD S=LIST(HD S), S= TL S	1001
END , S= LIST(X)	1002
END	1003
END	1004
ELSE S= X,	1005
NEGS= NIL,	1006
WHILE S REPEAT DO	1007
NEGS= GENCLAUS(GENSYM(), NIL, HD S) :NEGS, S= TL S	1008
END,	1009
CLEARSYS(),	1010
CLEARSYS=NIL,	1011
GARBCOLL(),	1012
COPY=COPX,	1013
PRINT(*START OF RUN*, STIME=TIME()),	1014
S= RTM(NEGS, MIXEDCL, COERS),	1015
PRINTHIS(S), PRINT(TIME()-STIME), PRINT(COPYFAIL), PRINT(*EOJ*),	1016
PRINT(*RESOLUTIONS*, RCNTR, *RESOLVENTS GENERATED*, RGEN, *ACCEPTED	1017
*, RACPT)	1018
,	1019
PRINT(*DUPLICATES*, DCNTR, *COMPLEX*, CNTR, *DEEP*, DEEPCNTR),	1020
STOP()	1021
END	1022
END;	1023
CHGCHAR(=., 10);	1024
CLEARSYS=PROC(),	1025
	1026

BEGIN(),	1027
GARBCOLL(),	1028
BALM= EXECUTE= VFROML.=NIL,	1029
ORDINAL.=ORD1.=IFROMID.=INITIAT.=INITIQ.=INITUNA.=NIL,	1030
UNARYLI.=INITINF.=INFIXLI.=INITEXP.=MACROLI.=NIL,	1031
INITCOD.=CONGENL.=INITUPL.=OPLIST.=INITMIS.=NIL,	1032
READ.=RDTOKEN.=RDTITEM.=GETLIST.=GETVECT.=GETV.=NIL,	1033
LXSCAN.=READIN.=TRANSLA.=MACDEF.=MACROLI.=INFIX=UNARY=NIL,	1034
BRACKET= FNOTN.=GETNEXT.=ANALYSE.=REMCOM.=REMSEP.=NIL,	1035
OPERROR.=EXPAND.=EXLIS.=CODEGEN.=SUBLBLS.=NIL,	1036
COMP.= GVAR.= GCON.=GCALLS.= ARGLIST.= GREFS.= ASS.=LBL.=	1037
GENLBL.=	1038
GLAMBDA=EXCH1.=EXCH2.= GPROP=SAVL1.=SAVL2.=RESTL1.=RESTL2.=NIL,	1039
GRETURN= GPROGN=GGQ=GCOND= GELSE.=GTHEN.=GAND=GOR=NIL,	1040
GQUOTE=GSETQ=ASSIGN.=GWHILE=GFOR=CGCHECK.= MAKPROPS=PRTP.=NIL,	1041
SETPROPL(= INFIX,NIL),SETPROPL(=UNARY,NIL),	1042
SETPROPL(=MACRO,NIL), SETPROPL(=CGDEG,NIL),SETPROPL(=INSTR,NIL)	1043
,	1044
SETPROPL(=INFIXLI.,NIL),SETPROPL(=UNARYLI.,NIL),	1045
SETPROPL(=MACROLI.,NIL),SETPROPL(=CODGENL.,NIL)	1046
, SETPROPL(=OPLIST.,NIL), REMOVEX.=MMEANS=MMATCH.=TMAC.=MATCH.=	1047
NIL,	1048
BUILD.=GENNAM.=AODON=CODEGEN=COMPILE=IFROMID=MACDEF=ORDINAL=NIL	1049
,	1050
RDTOKEN=READ=TRANSLAT= MGO.=MEQUAL.=NIL,	1051
CHAR.=NIL,	1052
END	1053
END;	1054
CHGCHAR(=.,11);	1055
RTM= PROC(NEGS,MIXEDCL,COERS),	1056
BEGIN(FINISHED,N,CURCONTE,NEGLIT,NEGCL,M,USET,CL,REDO,MSET,M,	1057
RETAIN,	1058
NUMSOLUT,UNIFOEPT),	1059
DETAIL(NEGS),DETAIL(FINISHED),	1060
WHILE =(N.NEGS)=NEGS REPEAT DO	1061
DETAIL(N), DETAIL(DONE),	1062
IF SAVER THEN DO	1063
IF SAVER(N) THEN DONE = N: DONE	1064
END ELSE DONE= N:DONE,	1065
NEGCL=LITLIST(N), NEGLIT=SELECT1(NEGCL), REDO= FALSE,	1066
DETAIL(NEGCL),DETAIL(NEGLIT),	1067
NEGCL=COPY(NEGCL),	1068
IF NEGPRINT THEN	1069
PRINT(=NEG CLAUSE IS#,NAMEOF(N),#LITERAL IS#,NEGLIT),	1070
NUMSOLUT=0,	1071
RCNTR= RCNTR+1,	1072
MSET=LOOKUP(HD NEGLIT,PREDCHA1),	1073
WHILE =(M.MSET)=MSET REPEAT DO	1074
FINISHED=MAXLITS-CLLENGTH(M)-CLLENGTH(N)+1,	1075
IF FINISHED GE 0 THEN DO	1076
POS= POSLIT OF(NAMEOF(M)),	1077
DETAIL(N),DETAIL(M),DETAIL(POS),	1078
UNIFOEPT=0,	1079
PENDING=NIL,CORID=100,	1080

```

BYPASS= NIL, 1081
IF((USET=MGUSET(COPY(NEGLIT),PDS,0,FINISHED,NIL)) NE = 1082
NOGOOD) THEN DO 1083
    IF CLAUSEPR THEN 1084
        PRINT(RESOLVE WITH,NAMEOF(M)), 1085
        WHILE =(U.USET)= USET REPEAT DO 1086
            DETAIL(U),DETAIL(USET), 1087
            RETAIN= TRUE, CL= GENNEG(N,M,COPY(NEGLIT),U, 1088
            NEGCL), 1089
            DETAIL(CL), 1090
            RGEN=RGEN+1, 1091
            RETAIN= DELATE(CL, RETAIN), 1092
            IF EMPTYCLA(CL) THEN RETURN CL ELSE DO 1093
                IF RETAIN THEN(IF CLMEMBER(CL,DONE) THEN 1094
                RETAIN= FALSE ELSE RETAIN= 1095
                ~CLMEMBER(CL,NEGS)), 1096
                IF RETAIN THEN ADDNEGS(CL) 1097
                , IF CLAUSEPR AND RETAIN THEN DO 1098
                    PRINT(RESOLVENT IS,NAMEOF(CL),LITLIST(CL 1099
                    )), 1100
                    PRINT(MERIT VALUE IS,MERITVAL(CL)), 1101
                    PRINT(POOL SIZE IS ,LENGTH(NEGS)) 1102
                END 1103
            END 1104
        END 1105
    END 1106
END 1107
END, 1108
IF (HD NEGLIT EQ EQ1) OR (HD NEGLIT EQ EQ2) THEN DO 1109
    RCNTR=RCNTR+1, 1110
    BYPASS= (HD NEGLIT EQ EQ2), 1111
    M=REFLEX, 1112
    FINISHED=MAXLITS-LENGTH(N)+1, 1113
    DETAIL(FINISHED) , 1114
    UNIFDEPT=1,PENDING=NIL,CORID=100, 1115
    IF(USET=MGUSET(COPY(HD TL NEGLIT),COPY(HD TL TL NEGLIT),0 1116
    ,FINISHED,NIL) 1117
    ) NE =NOGOOD THEN DO 1118
        DETAIL(USET), 1119
        IF CLAUSEPR THEN 1120
            PRINT(RESOLVE WITH,NAMEOF(M)), 1121
            WHILE =(U.USET)=USET REPEAT DO 1122
                DETAIL(U), 1123
                RETAIN=TRUE,CL=GENNEG(N,M,COPY(NEGLIT),U,NEGCL), 1124
                RGEN=RGEN+1, 1125
                RETAIN=DELATE(CL,RETAIN), 1126
                IF EMPTYCLA(CL) THEN RETURN(CL)ELSE DO 1127
                    IF RETAIN THEN (IF CLMEMBER(CL,DONE) THEN RETAIN 1128
                    = FALSE ELSE 1129
                    RETAIN=~CLMEMBER(CL,NEGS)), 1130
                    IF RETAIN THEN ADDNEGS(CL) 1131
                    , IF CLAUSEPR AND RETAIN THEN DO 1132
                        IF CLAUSEPR THEN 1133
                        PRINT(RESOLVENT IS,NAMEOF(CL),LITLIST(CL)), 1134

```


PRINT(*POOL SIZE IS *,LENGTH(NEGS))	1135
END	1136
END	1137
END	1138
END,	1139
IF (HD NEGLIT EQ EQ1) OR (HD NEGLIT EQ EQ2) THEN DO	1140
BYPASS= NIL,	1141
M= TRANS,	1142
FINISHED=MAXLITS-LENGTH(N)-1,	1143
IF FINISHED GE 0 THEN DO	1144
IF CLAUSEPR THEN	1145
U= NIL:LIST(LIST(EQ2, COPY(HD TL NEGLIT),=ZZZ),	1146
LIST(EQ2, =ZZZ,COPY(HD TL TL NEGLIT))),	1147
PRINT(*RESOLVE WITH*,NAMEOF(M)),	1148
RETAIN=TRUE,	1149
CL= GENNEG(N,M,COPY(NEGLIT),L,NEGCL),	1150
RGEN= RGEN+1,	1151
IF CLAUSEPR THEN	1152
PRINT(*RESOLVENT IS*,NAMEOF(CL),LITLIST(CL)),	1153
RETAIN= DELATE(CL,RETAIN),	1154
IF CLAUSEPR THEN	1155
PRINT(*POOL SIZE IS*,LENGTH(NEGS)),	1156
IF RETAIN THEN (IF CLMEMBER(CL,DONE) THEN RETAIN=	1157
FALSE ELSE RETAIN=	1158
¬CLMEMBER(CL,NEGS)),	1159
IF RETAIN THEN ADDNEGS(CL)	1160
END	1161
END,	1162
IF REDD THEN ADDNEGS(N)	1163
END	1164
END	1165
END;	1166
COMPLEX= PROC(X),	1167
0	1168
END;	1169
ONCE= PROC(LIT),	1170
BEGIN(J,K),	1171
J=NAMEOF(MIXEDPAR(CURCONTE)),K=NEGPARG(CURCONTE),	1172
WHILE(¬INPUTCL(K)) REPEAT DO	1173
IF(NAMEOF(MIXEDPAR(K)) EQ J) THEN RETURN FALSE,	1174
K= NEGPARG(K)	1175
END,	1176
RETURN TRUE	1177
END	1178
END;	1179
ONCE PARTEVAL ONCE;	1180
UNIQ = PROC(N,P,UOUT),	1181
BEGIN(),	1182
RETURN UOUT EQ NIL	1183
END	1184
END;	1185
SYM1= PROC(N,P,UOUT),	1186
BEGIN(),	1187
	1188

IF GROUND(N) OR GROUND(P) THEN RETURN	1189
TRUE ELSE RETURN FALSE	1190
END	1191
END;	1192
DOWN3OP =PROC(PRED),	1193
BEGIN(),	1194
MERIT=MERIT+30, RETURN TRUE	1195
END	1196
END;	1197
DOWN3O PARTEVAL DOWN3OP;	1198
PREDCHAI= NIL;	1199
GENCLAUS(=REFLEX,=(EQ1 X X), NIL);	1200
GENCLAUS(=TRANS,NIL,NIL);	1201
SAVER=NIL;	1202
SELECT1= PROC(LITS),	1203
BEGIN(Y),	1204
Y=LITS, WHILE Y REPEAT DO	1205
IF HD HD Y EQ =MARK THEN Y= TL Y ELSEIF	1206
(GETPROP(HD HD Y ,=PE) NE NIL) THEN Y= TL Y ELSE RETURN HD Y	1207
END,	1208
PRINT(=NO LITERAL MAY BE SELCTED*,LITS),	1209
RETURN HD Y	1210
END	1211
END;	1212
COERS=NIL;	1213
RCNTR= RGEN= RACCPT =0;	1214
NOTEQLIS=NIL;	1215
COMPLET = PEFNS= NIL;	1216
DONE=NIL;	1217
MIXEDCL= NIL;	1218
COPYFAIL= 0;	1219
BVARS= TRUE;	1220
CLAUSEPR= TRUE;	1221
CORID= 0;	1222
DEBUG=FALSE;	1223
EQ1= =EQ1; EQ2= =EQ2;	1224
CLAUSEID=1;	1225
DONE=NIL;MAXLEVEL=10;	1226
BYPASS= NIL; MAXCOERS=1; NOTEQLIS=NIL;	1227
MAXCPLEX = 6;	1228
MAXCPX= 65;	1229
MAXNEST=7;	1230
NEGPRINT = TRUE;	1231
NOSIMPLE = FALSE;	1232
TRACE=1;	1233
MAXLEVEL=100;	1234
MAXLITS=10;	1235
DO	1236
SAVEBALM(=DILEMMA=),PRINT(=DILEMMA,=JUNE,76)	1237
END;	1238
CLOSE(=DILEMMA=);	1239

*	1
*	2
*	3
*	4
*	5
*	6
ONEWAY= TRUE;	7
ALWAYS= FALSE;	8
CONEF=IFROMID;	9
MAXNEST=5;	10
MAXLEVEL=15;	11
MAXFREE=4;	12
MAXCPX=120;	13
CONSTANT	14
STANG, RTANG, R	15
END;	16
CONSTANT	17
I	18
END;	19
CONSTANT	20
E,P,K,N,M,C,D,Q,B	21
END;	22
COEF=CONEF;	23
MAXCOERS=1;	24
POINTQ= PROC(X),	25
MEMBER(X,=(E P K N M C D Q B))	26
END;	27
TYPE XPT = POINTQ;	28
TYPE YPT= POINTQ;	29
TYPE ZPT= POINTQ;	30
TYPE UPT= POINTQ;	31
TYPE VPT= POINTQ;	32
TYPE WPT = POINTQ;	33
TYPEQ= PROC(X) ,	34
MEMBER(X,=(R I))	35
END;	36
RQ= PROC(X),	37
X EQ =R	38
END; BQ= PROC(X),	39
X EQ =I	40
END;	41
TYPE TYR = RQ; TYPE TYB = BQ;	42
*	43
* SPECIAL CPROCS OF GEOMETRY	44
*	45
CSYMP= PROC(N,P,UOUT),	46
UOUT EQ NIL	47
END;	48
LFIPP= PROC(N,P,UOUT),	49
BEGIN(),	50
WORKING =	51
(52
IF VARIABLE(N) THEN FALSE ELSE IF VARIABLE(P) THEN FALSE ELSE	53
IF GROUND(N) THEN TRUE ELSE IF GROUND(P) THEN TRUE ELSE FALSE	54

```

    )
    , IF DEBUG THEN PRINT(WORKING), RETURN WORKING
END
END;
LFLIPP CPROC LFLIP;
*
* PARTIAL EVALUATORS OF FUNCTIONS AND PREDICATES
* CANONICAL ORDERINGS ARE DONE HERE
*
RLINE= PROC(FN),
BEGIN(X,Y),
    CRACK(=(T1 X Y),FN),
    IF VARIABLE(X) THEN DO
        IF VARIABLE(Y) THEN DO
            MERIT=MERIT+10,
            IF X LT Y THEN RETURN FN,
            IF X EQ Y THEN DO
                RETAIN = FALSE, RETURN FN
            END,
            RETURN LIST(T1,Y,X)
        END,
        RETURN FN
    END,
    IF VARIABLE(Y) THEN RETURN LIST(T1,Y,X),
    IF(HD X EQ =INTER) THEN RETURN FN,
    IF(HD Y EQ =INTER) THEN RETURN FN,
    XPT= HD X, YPT= HD Y,
    IF XPT EQ YPT THEN DO
        RETAIN= FALSE, RETURN FN
    END,
    IF CONEF(XPT) LE CONEF(YPT) THEN RETURN FN,
    RETURN LIST(T1,LIST(YPT),LIST(XPT))
END
END;
RANG= PROC(FN),
BEGIN(T1,X,Y,Z),
    CRACK(=(T1 X Y Z),FN),
    IF EQUAL(X,Y) THEN RETAIN= FALSE,
    IF EQUAL(X,Z) THEN RETAIN= FALSE,
    IF EQUAL(Y,Z) THEN RETAIN= FALSE,
    IF ~CRACK(=(T1 XPT Y ZPT),FN) THEN RETURN FN,
    IF ( COLINP(FN) EQ TRUE) THEN RETURN LIST(=STANG),
    IF CONEF(XPT) LE CONEF(ZPT) THEN RETURN FN,
    IF ~PAIRQ(Y) THEN
        IF ~VARIABLE(Y) THEN Y= LIST(Y),
    RETURN LIST(T1,LIST(ZPT),Y,LIST(XPT))
END
END;
L PARTEVAL RLINE; A PARTEVAL RANG;
MIDPTP= PROC(FN),
BEGIN(X,Y,T1),
    CRACK(=(T1 X Y),FN), IF EQUAL( X,Y) THEN RETAIN= FALSE,
    RETURN FN
END

```

55
56
57
58
59
60
61
62
63
64
65
66
67
68
69
70
71
72
73
74
75
76
77
78
79
80
81
82
83
84
85
86
87
88
89
90
91
92
93
94
95
96
97
98
99
100
101
102
103
104
105
106
107
108

END;	109
MIDPT PARTEVAL MIDPTP;	110
DIFF3= PROC(PRED),	111
BEGIN(),	112
CRACK(=(T1 XPT1 YPT1 ZPT1),PRED) ,	113
IF EQUAL(XPT1,YPT1) THEN RETURN FALSE,	114
IF EQUAL (XPT1,ZPT1) THEN RETURN FALSE,	115
IF EQUAL(ZPT1,YPT1) THEN RETURN FALSE,	116
IF CRACK(=(T1 XPT YPT ZPT),PRED) THEN RETURN TRUE , THEN DO	117
IF ~ MEMBER(XPT,LIST(ZPT,UPT,VPT,WPT)) AND (~MEMBER(YPT,	119
LIST(120
ZPT,UPT,VPT,WPT))) THEN RETURN TRUE, RETURN FALSE	121
END,	122
RETURN PRED	123
END	124
END;	125
DIFF3 PARTEVAL DIFF3;	126
DOWN1 = PROC(PRED),	127
BEGIN(),	128
IF ~CRACK(=(T1 XPT YPT ZPT UPT VPT WPT),PRED) THEN MERIT=MERIT+	129
30,	130
RETURN TRUE	131
END	132
END;	133
DOWN PARTEVAL DOWN1;	134
COLINP= PROC(FN),	135
BEGIN(Z,S1,S2),	136
IF CRACK(=(T1 XPT YPT ZPT), FN) THEN DO	137
IF MEMB(TL FN,COLINER) THEN RETURN TRUE,	138
RETURN FALSE	139
END,	140
IF CRACK(=(T1 XPT YPT Z),FN) THEN DO	141
IF VARIABLE(Z) THEN DO	142
BIND(Z,LIST(=INTER,=(R),CORID =CORID+1,LIST(XPT),	143
LIST(YPT),CORID=CORID+1,CORID=CORID+1)),	144
RETURN TRUE	145
END,	146
IF CRACK(=(T1 TYR T1 UPT VPT S1 S2),Z) THEN DO	147
IF (XPT EQ UPT) AND (YPT EQ VPT) THEN RETURN TRUE,	148
IF VARIABLE(S1) AND VARIABLE(S2) THEN DO	149
BIND(S1,XPT),BIND(S2,YPT),BIND(T1,=(R)),RETURN TRUE	150
END,	151
IF ~ CROSSES(LIST(=INTER,XPT,YPT,UPT,VPT))THEN RETURN	152
FALSE,	153
RETURN TRUE	154
END	155
END,	156
IF CRACK(=(T1 Z XPT YPT),FN) THEN RETURN COLINP(LIST(T1,LIST(157
YPT),	158
LIST(XPT),Z)),	159
RETURN FN	160
END	161
END;	162

COLIN PARTEVAL COLINP;	163
CROSSES= PROC(PRED),	164
BEGIN(T,L,S,J),	165
IF ¬ CRACK(=(T XPT YPT UPT VPT),	166
PRED) THEN RETURN PRED,	167
IF COEF(XPT) GT COEF(YPT) THEN DO	168
T= XPT,XPT= YPT, YPT= T	169
END,	170
IF COEF(UPT) GT COEF(VPT) THEN DO	171
T= UPT, UPT= VPT, VPT= T	172
END, S= LIST(XPT,YPT), J= LIST(UPT,VPT),	173
L= LOOKSP, WHILE L REPEAT DO	174
IF MEMB(S, HD L) THEN RETURN ¬MEMB (J,HD L), L= TL L	175
END,	176
RETURN TRUE	177
END	178
END ; CROSSES PARTEVAL CROSSES;	179
INTER= PROC(FN),	180
BEGIN(T,S1,S2,S3,S4,TY1,TY2,REP,ITEM),	181
IF CRACK(=(T TY1 TY2 XPT YPT UPT VPT),FN) THEN DO	182
IF CONEF(XPT) GT CONEF(UPT) THEN DO	183
T= XPT, XPT= UPT, UPT= T, T= VPT, VPT=YPT,YPT=T	184
END,	185
S1=PNAMES,S2=LIST(XPT,YPT,UPT,VPT),	186
WHILE S1 REPEAT DO	187
ITEM=HD HD S1, REP= HD TL HD S1,	188
IF EQUAL(S2, TL TL ITEM) THEN DO	189
IF(TY1 EQ HD ITEM) THEN DO	190
IF(TY2 EQ HD TL ITEM) THEN RETURN REP,	191
IF VARIABLE(TY2) THEN DO	192
BIND(TY2,HD TL ITEM), RETURN REP	193
END	194
END,	195
IF(TY2 EQ HD TL ITEM) THEN DO	196
IF VARIABLE(TY1) THEN DO	197
BIND(TY1,HD ITEM), RETURN REP	198
END	199
END,	200
IF VARIABLE(TY1) AND VARIABLE(TY2) THEN DO	201
BIND(TY1,HD ITEM),BIND(TY2,HD TL ITEM),RETURN REP	202
END	203
END,	204
S1= TL S1	205
END	206
END,	207
IF CRACK(=(T TYR TYR S1 XPT S2 YPT),FN) THEN (IF XPT EQ YPT	208
THEN RETURN	209
YPT),	210
IF CRACK(=(T TYB TYR XPT S1 S2 YPT),FN) THEN (IF XPT EQ YPT	211
THEN	212
RETURN XPT),	213
IF CRACK(=(T TYB TYR S1 XPT S2 YPT) ,FN) THEN(IF XPT EQ YPT	214
THEN	215
RETURN XPT),	216

IF CRACK(=(T TYB TYB S1 S2 S3 S4),FN) THEN DO	217
IF PAIRO(S1) THEN S1= HD S1, IF PAIRO(S2) THEN S2= HD S2,	218
IF PAIRO(S3) THEN S3= HD S3, IF PAIRO(S4) THEN S4= HD S4,	219
IF S1 EQ S3 THEN RETURN S1, IF S1= S4 THEN RETURN S1,	220
IF S2 EQ S3 THEN RETURN S2, IF S2 EQ S4 THEN RETURN S2	221
END,	222
RETURN FN	223
END	224
END;	225
INTER PARTEVAL INTER;	226
*	227
* DILEMMA ASSERTIONS AND THEOREMS	228
*	229
* ANGLE EQUALITY	230
ASYM COER EQANG(X,Y),EQANG(Y,X);	231
ANAME COER A(X,Y,Z),A(S,Y,R),COLIN(X,S,Y) ^ COLIN(Z,R,Y);	232
ANAME1 COER A(X,Y,Z),A(S,Y,Z),COLIN(X,S,Y);	233
ANAME2 COER A(X,Y,Z),A(X,Y,R),COLIN(Z,R,Y);	234
VERTICAL THEOREM EQANG(A(X,Y,Z),A(U,Y,W)) FROM COLIN(X,Y,W) ^ COLIN(U	235
,Y	236
,Z)	237
;	238
ANGR COER A(X,Y,Z),A(Z,Y,X);	239
UNIQ CPROC ANAME1; UNIQ CPROC ANAME2; UNIQ CPROC ANAME;	240
FLIPP CPROC ANGR;	241
ALTER2 THEOREM EQANG(A(Z,Y,X),A(Z,R,S)) FROM PARALLEL(L(X,Y),L(R,S))	242
^ COLIN(R,Z,Y);	243
ALTER1 THEOREM EQANG(A(X,Y,Z),A(U,Z,Y)) FROM PARALLEL(L(Y,X),L(U,Z));	244
AREFLX ASSERT EQANG(X,X);	245
ATRANS THEOREM EQANG(X,Y) FROM EQANG(X,Z) ^ EQANG(Z,Y) ^ DOWN30(X);	246
APART THEOREM EQANG(A(X,U,Z),A(V,R,T)) FROM EQANG(A(X,U,Y),A(V,R,W))	247
^ EQANG(A(Y,U,Z),A(W,R,T)) ^ DOWN30(X);	248
ACONG THEOREM EQANG(A(X,Y,Z),A(U,V,W)) FROM CONG(X,Y,Z,U,V,W)	249
^ DIFF3(X,Y,Z) ^ DIFF3(U,V,W);	250
ACOR THEOREM EQANG(A(R,Y,X),A(Z,X,J)) FROM PARALLEL(L(W,R),L(U,Z))	251
^ COLIN(W,Y,R) ^ COLIN(U,X,Z) ^ COLIN(Y,X,J);	252
AISSOC THEOREM EQANG(A(X,Y,Z),A(Y,X,Z)) FROM EQLINE(L(X,Z),L(Y,Z));	253
ATRANS THEOREM EQANG(X,Y) FROM EQANG(X,Z) ^ EQANG(Z,Y) ^ DOWN30(X);	254
RTANG THEOREM EQANG(A(X,Y,Z),RTANG) FROM EQANG(A(T,Y,X),RTANG)	255
^ COLIN(T,Y,Z);	256
SEGMENT EQUALITY	257
FLIP COER L(X,Y),L(Y,X);	258
SYM COER EQLINE(X,Y),EQLINE(Y,X);	259
PARBASE THEOREM EQLINE(L(X,Y),L(Y,Z)) FROM	260
EQLINE(L(Z,W),L(W,V)) ^ PARALLEL(L(Y,W),L(X,V)) ^ COLIN(X,Y,Z) ^	261
COLIN(Z,W,V) ^ ONCE(X);	262
CONG THEOREM EQLINE(L(X,Y),L(U,V)) FROM CONG(Y,Z,X,V,W,U) ^ DIFF3(X,	263
,	264
)	265
^ DIFF3(U,V,W);	266
YMI CPROC LSYM; SYMI CPROC ASYM;	267
MID ASSERT EQLINE(L(X,MIDPT(X,Y)),L(Y,MIDPT(X,Y)));	268
PART THEOREM EQLINE(L(X,Y),L(U,V)) FROM	269
EQLINE(L(X,R),L(U,S)) ^ EQLINE(L(R,Y),L(S,V)) ^ COLIN(X,R,Y) ^	270

COLIN(U,	271
S,V);	272
LTRANS THEOREM EQLINE(L(X,Y),L(U,V)) FROM EQLINE(L(X,Y),L(S,T)) ^	273
EQLINE(L(S,T),L(U,V)) ^ DOWN30(X);	274
LREF ASSERT EQLINE(X,X);	275
LISOC THEOREM EQLINE(L(X,Z),L(Z,Y)) FROM EQANG(A(Z,X,Y),A(X,Y,Z));	276
* PARALLEL LINE THEOREMS	277
PARTRANS THEOREM	278
PARALLEL(X,Y) FROM PARALLEL(X,Z) ^ PARALLEL(Z,Y) ;	279
PCR1 THEOREM PARALLEL(L(X,INTER(TP1,TP2,U,V,X,Z)),L(X,Z))	280
FROM CROSSES(X,Z,U,V);	281
PARINT THEOREM PARALLEL(L(INTER(T1,T2,X,Y,U,V),V),L(S,T)) FROM	282
PARALLEL(L(S,T),L(U,V));	283
MIDPAR THEOREM	284
PARALLEL(L(Y,U),L(Z,V)) FROM	285
EQLINE(L(Z,Y),L(Y,X)) ^ COLIN(Z,Y,X) ^ EQLINE(L(V,U),L(U,X)) ^	286
COLIN(V,U,X) ^ ONCE(X);	287
PCR2 THEOREM PARALLEL(L(X,INTER(TP1,TP2,X,Z,U,V)),L(X,Z)) FROM CROSSES	288
(X,Z,U,V);	289
PEND THEOREM PARALLEL(L(X,Y),L(X,Z)) FROM COLIN(X,Y,Z);	290
PRASE THEOREM PARALLEL(L(X,Y),L(U,V)) FROM	291
PARALLEL(L(S,Y),L(U,V)) ^ COLIN(X,S,Y);	292
PEXT THEOREM PARALLEL(L(X,Y),L(U,V)) FROM	293
PARALLEL(L(X,Y),L(R,V)) ^ COLIN(R,U,V);	294
PSYM COER PARALLEL(X,Y), PARALLEL(Y,X);	295
P1 THEOREM PARALLEL(L(X,Y),L(U,V)) FROM EQANG(A(Y,Z,W),A(V,W,R)) ^	296
COLIN(X,Y,Y) ^ COLIN(Y,W,V) ^ COLIN(S,Z,W,R);	297
P2 THEOREM PARALLEL(L(Z,Y),L(W,V)) FROM	298
EQANG(A(Y,Z,W),A(V,W,S)) ^ COLIN(Z,W,S);	299
* TRIANGLE CONGRUENCE SECTION	300
CSYM COER CONG(X,Y,Z,U,V,W),CONG(U,V,W,X,Y,Z);	301
CASA THEOREM CONG(X,Y,Z,U,V,W) FROM	302
EQLINE(L(U,V),L(X,Y)) ^ EQANG(A(X,Y,Z),A(U,V,W)) ^ EQANG(A(Y,X,Z),	303
A(W,U,V));	304
*	305
CSYMP CPROC CSYM;	306
CSSS THEOREM CONG(X,Y,Z,U,V,W) FROM	307
EQLINE(L(X,Y),L(U,V)) ^ EQLINE(L(Y,Z),L(V,W)) ^ EQLINE(L(X,Z),L(U,W	308
));	309
CSAS THEOREM CONG(X,Y,Z,U,V,W) FROM	310
EQLINE(L(X,Y),L(U,V)) ^ EQANG(A(X,Y,Z),A(U,V,W)) ^ EQLINE(L(Y,Z),L(311
V,W)	312
);	313
CPERM THEOREM CONG(X,Y,Z,U,V,W) FROM CONG(Y,Z,X,V,W,U);	314
CFLIP THEOREM CONG(X,Y,Z,U,V,W) FROM CONG(X,Z,Y,U,W,V);	315
CPARAL THEOREM CONG(X,Y,Z,X,U,Z) FROM PARALLGM(X,U,Z,Y);	316
CTrans THEOREM CONG(X,Y,Z,U,V,W) FROM CONG(X,Y,Z,P,Q,R) ^ CONG(P,	317
Q,R,U,V,W) ^ DOWN(X,Y,Z,U,V,W);	318
CREF ASSERT CONG(X,Y,Z,X,Y,Z);	319
*	320
* COMPLEXITY MEASURE AND SELECTION FUNCTIONS	321
* THIS SECTION HOLDS THE HEURISTIC INFORMATION	322
*	323
GRNDUNIT=NIL;	324

SELECT1= PROC(LITS),	325
BEGIN(Y,A),	326
Y=LITS,	327
DETAIL(Y),	328
WHILE Y REPEAT DO	329
IF HD HD Y EQ =PARALLEL THEN	330
DO	331
A=HD TL HD Y,	332
IF PAIRQ(A) THEN A= HD TL TL A,	333
IF PAIRQ(A) THEN A= HD A,	334
DETAIL(A),	335
IF A EQ =INTER THEN RETURN HD Y	336
END,	337
* TEST IF HD Y IS (PARALLEL (L X (INTER ...)) ())	338
Y= TL Y	339
END,	340
Y= LITS, WHILE Y REPEAT DO	341
DETAIL(Y),	342
IF HD HD Y EQ =MARK THEN Y= TL Y	343
ELSEIF GETPROP(HD HD Y ,=PE) THEN Y= TL Y ELSE RETURN HD Y	344
END,	345
PRINT(=NO,=LIT,=MAY,=BE,=SELECTED,LITS),	346
RETURN HD LITS	347
END	348
END;	349
COMPLEX= PROC(LITS),	350
BEGIN(S,J,FND,HTMP,FAIL,PCNT,CNT,CODE),	351
IF ~ LITS THEN RETURN 0,	352
IF FREEVARS(LITS) GT MAXFREE THEN RETURN 1000,	353
CODE=TRUE,	354
IF LENGTH (LITS) EQ 1 THEN GRNSAVE= HD LITS,	355
PCNT=0,CNT=0,	356
S=0,	357
FAIL= TRUE,	358
IF LENGTH(LITS) LT (MAXLITS/2) THEN FAIL= FALSE,	359
IF ~INPUTCL(CURCONTE) THEN DO	360
J=NAMEOF(MIXEDPAR(CURCONTE)),	361
IF MEMBER(J,HYP) THEN S= S-50	362
,	363
IF DEPTHVAL(CURCONTE) GT 6 THEN DO	364
J= NAMEOF(MIXEDPAR(NEGPARG(CURCONTE))), IF MEMBER(J,HYP)	365
THEN S= S-	366
50 ELSEIF MEMBER(NAMEOF(MIXEDPAR(NEGPARG(NEGPARG(CURCONTE)	367
))),HYP) THEN	368
S=S-30	369
END	370
END,	371
WHILE LITS REPEAT DO	372
IF ~((J=HD HD LITS) EQ =MARK) THEN DO	373
IF J EQ =PARALLEL THEN PCNT= PCNT +1,	374
IF J EQ =COLIN THEN CNT= CNT +1,	375
IF ~(GETPROP(J,=PE)) THEN DO	376
IF GROUND(HD LITS) THEN DO	377
IF MEMBER(HD LITS, GRNDUNIT) THEN DO	378

```

PRINT(=SUBSUMED,=BY,=GROUND,=NEGATIVE,=UNIT), 379
RETAIN= FALSE,RETURN 1000 380
END 381
END, 382
CODE=FALSE, 383
IF HASCON(HD LITS) THEN DO 384
FND=FALSE,HTEMP=HYP, 385
WHILE(¬FND) AND HTEMP REPEAT DO 386
FND= (MGUSET(POSLIT OF (HD HTEMP),COPY(HD LITS), 387
0,0, 388
NIL) NE 389
¬NOGOOD), IF FND THEN FAIL=FALSE, 390
HTEMP= TL HTEMP 391
END, 392
IF FND THEN S= S-20 ELSE S= S+10 393
END , DO 394
S= S+2*VARSNUM(HD LITS), 395
IF J EQ =PARALLEL THEN FAIL=FALSE, 396
IF J EQ =CONG THEN DO 397
S= S+ 1, FAIL= FALSE 398
END ELSE S= S+1, 399
IF ¬GROUND(HD LITS) THEN S= 400
S+ 5 401
END 402
END 403
ELSE DO 404
S= S+4*VARSNUM(HD LITS) +5 405
END 406
END , LITS = TL LITS 407
END, 408
IF PCNT GE 3 THEN RETURN 1000, 409
IF CODE THEN DO 410
PRINT(=POSSIBLE,=NULL,=CLAUSE),RETURN 1000 411
END, 412
IF FAIL THEN RETURN 1000, 413
IF CLENGTH(CURCONTE) EQ 1 THEN DO 414
IF GROUND(GRNSAVE) THEN GRNDUNIT= COPY(GRNSAVE):GRNDUNIT 415
END, 416
RETURN S 417
END 418
END; 419
* 420
* THE INPUT FOR THE G5.1 EXAMPLE 421
* 422
* G5 EXAMPLE 6 423
HYP= =(H1 H2 H3); 424
COLINER = =(((E)(P)(K)) ( (K)(P)(E) )((P)(N)(M)) ((M)(N)(P)) 425
((K)(M)(D)) ((D)(M)(K)) 426
( (E)(B)(D)) ((D)(B)(E)) 427
((C)(M)(B))((B )(M)(C)) 428
((E)(N)(C)) ((C)(N)(E)) ); 429
H1 ASSERT PARALLEL(L(K,C),L(E,D)); 430
H2 ASSERT EQLINE(L(E,N),L(N,C)); 431
H3 ASSERT EQLINE(L(K,M),L(M,D)); 432

```

H4 ASSERT PARALLEL(L(P,M),L(N,M));	433
HYP = =(H1 H2 H3 H4);	434
LOOKSP=((K C)(P N) (N M) (P M) (E D));	435
P NAMES=(((R R K P D E) E) ((R R C N D E) E) ((I R E K M N) P)	436
((I I E C P M) N) ((R R E D K M) D));	437
MEMBER= MEMB;	438
PROVE(EQLINE(L(E,P),L(P,K)));	439

Bibliography

- [1] Anderson, R., "Completeness Results for E-Resolution,"
Proc. AFIPS 1970 Spring Joint Computer Conf.,
pp. 653-656 (1970).
- [2] Andrews, P. B., "Resolution with Merging" JACM 15 (3),
pp. 367-381 (1968).
- [3] Black, F., "A Deductive Question Answering System,"
in Minsky (ed.) Semantic Information Processing,
pp. 354-402.
- [4] Chang, C. L., "The Unit Proof and the Input Proof in
Theorem Proving," JACM 17 (4) pp. 698-707 (1970).
- [5] Chang, C. L., "Theorem Proving with Variable-Constrained
Resolution," Inform. Sci. 1972 (4), pp. 217-231.
- [6] Coles, L. S., "An Online Question-Answering System with
Natural Language and Pictorial Input," Proc. Nat. ACM
Conf. 1968, pp. 157-167.
- [7] Darlington, J. L., "A Comit Program for the Davis-Putnam
Algorithm," Research Laboratory, Electron. Mech. Transla-
tion Group, M.I.T., May 1962.
- [8] Davis, M., and Putnam, H., "A Computing Procedure for
Quantification Theory," JACM 7 (3) pp. 201-215 (1960).
- [9] Dixon, J., "Z-Resolution: Theorem Proving with
Compiled Axioms," JACM 20 (1) pp. 127-147 (1973).
- [10] Gelernter, H., "Realization of a Geometry Theorem
Proving Machine," Proc. Intern. Conf. Inform. Processing
pp. 273-282 UNESCO House, Paris, 1959. In: Feigenbaum, E.,

- and Feldman, J. (eds.) Computers and Thought, pp. 134-152, McGraw-Hill, N. Y., 1963.
- [11] Gelernter, H., Hensen, J., and Loveland, D., Empirical Explorations of the Geometry Theorem Proving Machine," Proc. 1970 West. Joint Computer Conf., in: Feigenbaum, E., and Feldman, J. (eds.) Computers and Thought, pp. 153-167, McGraw-Hill, N. Y., 1963.
- [12] Gilmore, P., "A Procedure for the Productions from Axioms of Proofs for Theories Derivable within the First Order Predicate Calculus," Proc. IFIP Congr., 1959, pp. 265-273.
- [13] Goldstein, I., "Elementary Geometry Theorem Proving," M.I.T. AI Memo 280, April 1973.
- [14] Green, C., "Application of Theorem Proving to Problem Solving," Proc. IJCAI (1969) pp. 219-239.
- [15] Green, C., and Raphael, B., "The Use of Theorem Proving Techniques in Question-Answering Systems," Proc. 23 Nat. Conf. of the ACM (1968) pp. 169-181.
- [16] Harrison, M., Data Structures and Programming, Scott Foresman, 1973.
- [17] Harrison, M., and Brown, S., BALM: The BALM Programming Language, Courant Inst. of Math. Sci., NYU, 1974.
- [18] Harrison, M., and Rubin, N., "A Generalization of Resolution," New York Univ., 1976, submitted for publication.

- [19] Henschen, L., and Wos, L., "Unit Refutations and Horn Sets," JACM 21 (4), pp. 590-605 (1974).
- [20] Hewitt, C., "PLANNER: A Language for Manipulating Models and Proving Theorems in a Robot," Proc. IJCAI (1969).
- [21] Hewitt, C., Description and Theoretical Analysis (using Schemata) of PLANNER: A Language for Proving Theorems and Manipulating Models in a Robot, M.I.T. AI-TR-258, April 1972.
- [22] Horn, A., "On Sentences which are True of Direct Products of Algebras," J. Symbolic Logic 16, pp. 14-21 (1951).
- [23] Kowalski, R., and Kuehner, D., "Linear Resolution with Selector Function," Artificial Intelligence 2, pp. 227-260 (1971).
- [24] Loveland, D., "A Simplified Format for the Model Elimination Theorem Proving Procedure," JACM 16 (3), pp. 349-363 (1969).
- [25] McCarthy, J., "Programs with Common Sense," Proc. Symp. on Mechanization of Thought Processes, Her Majesty's Stationery Office, 1959, in: Minsky, M. (ed.) Semantic Information Processing, M.I.T. Press, 1968.
- [26] Minsky, M., "Descriptive Languages and Problem Solving," Western Joint Computer Conf., 1961, in: Minsky, M. (ed.), Semantic Information Processing, M.I.T. Press, 1968.
- [27] Morris, J., "E-Resolution: An Extension of Resolution to Include the Equality Relation," Proc. IJCAI, 1969.
- [28] Nevins, A., "Plane Geometry Theorem Proving Using Forward Chaining," M.I.T. AI Memo 303, Jan. 1974.

- [29] Raphael, B., "SIR: A Computer Program for Semantic Information Retrieval," in: Minsky, M. (ed.)
Semantic Information Processing, M.I.T. Press, 1968.
- [30] Robinson, J., "A Machine Oriented Logic Based on the Resolution Principle," JACM 12 (1) pp. 23-41, 1965.
- [31] Robinson, G., and Wos, L., "Axiom Systems in Automatic Theorem Proving," in: Landet (ed.) Proc. Symp. on Automatic Demonstration, Versailles, France, 1968, Springer-Verlag, (LNM-125), 1970.
- [32] Robinson, G., and Wos, L., "Paramodulation and Theorem Proving in First Order Theories with Equality," in: Meltzer, B., and Michie, D. (eds.), Machine Intelligence 4, American Elsevier, 1969.
- [33] Roboh, R., and Sacerdoti, E., "A Preliminary QLISP Manual," SRI Tech. Note 81, Aug. 1973.
- [34] Rulifson, J., "QA4 Programming Concepts," SRI Tech. Note, 60, Aug. 1971.
- [35] Rulifson, J., Waldinger, R., and Derksen, J.,
"A Language for Writing Problem-Solving Programs," SRI Tech. Note. 48, Oct. 1970.
- [36] Rulifson, J., Waldinger, R., and Perksen, J.,
"QA4 Working Paper," SRI Tech. Note 42, Oct. 1970.
- [37] Rulifson, J., "Preliminary Specifications of the QA4 Language," SRI Tech. Note 50, April 1970.
- [38] Rulifson, J., Waldinger, R., and Derksen, J.,
"A Language for Writing Problem Solving Programs," Proc. IFIP Congress, 1968.

- [39] Silbert, ., "A Machine Oriented Logic Incorporating the Equality Relation," in: Meltzer, B., and Michie, D., (eds.), Machine Intelligence 4, American Elsevier, 1969.
- [40] Stickel, M., "A Complete Unification Algorithm for Associative-Commutative Functions," 4th IJCAI, Tbilisi, Georgia, U.S.S.R., Sept. 1975.
- [41] Sussman, G., and McDermott, D., "From Planner to Conniver, A Genetic Approach," Proc. of FJCC, 1972.
- [42] Sussman, G., and Steele, G., "An Interpreter for Extended Lambda Calculus," M.I.T. AI Memo 349, Dec. 1975.
- [43] Sussman, G., Winograd, T., and Charniak, E., "MICRO-PLANNER Reference Manual," M.I.T. AI Memo 203A, Dec. 1971.
- [44] Winograd, T., "Procedures as a Representation for Data in a Computer Program for Understanding Natural Language," M.I.T., MAC-TR-84, Feb. 1971.
- [45] Yates, R., Raphael, B., and Hart, T., "Resolution Graphs," Artificial Intelligence 1 pp. 257-289, (1970).

DEC 21 1976

DATE DUE

RESERVE BOOK

GAYLORD

PRINTED IN U.S.A.

NYU NSO-10

c.2

Rubin

A hierarchical technique for...

**N.Y.U. Courant Institute of
Mathematical Sciences**

251 Mercer St.

New York, NY 10012

